**VŠB - Technická univerzita Ostrava**

**Fakulta elektrotechniky a informatiky**

**Katedra informatiky**

# Vývoj herních aplikací na platformách iOS a Android OS

Games Development on iOS and Android OS

2012 Lukáš Kuděla

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Lukáš Kuděla**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Vývoj herních aplikací na platformách iOS a Android OS
Games Development on iOS and Android OS

Zásady pro vypracování:

Vývoj a návrh herních aplikací určených pro mobilní zařízení s operačním systémem iOS a Android. Realizace prostřednictvím inkrementálně řízeného projektu. Návrh, analýza, design, rollout na prodejní platformu. Popis výhod/nevýhod jazyku LUA. Návrh a implementace herní logiky a interakce s 3D. Modelování ve 2D světě, interakce objektů hry s mapovými elementy. Vytváření modelu inteligence jednotlivých herních objektů. Vytvoření modelu vitality a práce se zdroji. Škálování inteligence podle nastavení obtížnosti hry. Implementace datového modelu hry a jejich objektů v prostředí cílového HW (iOS, Android OS). Transformace relačního modelu do cílového XML. Testování bude probíhat podle firemní metodiky.

1. Analýza aplikace prostřednictvím objektové metodologie postavené na UML specifikaci.
2. Návrh herní logiky a interakce s 3D.
3. Implementace v jazyce LUA - v prostředí Shiva 3D (společnost StoneTrip).
4. Verzování aplikace.
5. Testování, návrh testovacích scénářů.
6. Výstupem práce bude hra kombinující prvky 3D akce a strategie.

Seznam doporučené odborné literatury:

Sayed Hashimi, Pro Android 2, Apress, 2010, ISBN-13: 978-1430226598
Reto Meier, Professional Android 2 Application Development, Wrox, 2010, ISBN-13: 978-0470565520
David Mark, Beginning iPhone 4 Development: Exploring the iOS SDK, Apress; 1 edition, 2011, ISBN 978-1430230243

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Tomáš Ivanský**

Konzultant diplomové práce: Ing. Michal Krumnikl

Datum zadání: 18.11.2011

Datum odevzdání: 04.05.2012


doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských/magisterských programech VŠB-TU Ostrava.

Použité diagramy, zdrojový kód a grafika jsou vlastnictvím společnosti Zoongo, s.r.o. a mohou být šířeny pouze s jejím souhlasem.

V Ostravě dne 1.5.2012 .....

ZOONGO, s.r.o.
Mešnická 142
74285 Vřesina
IČO: 286 50 883
DIČ: CZ28650883

.....
podpis

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 1.5.2012 .....

.....
podpis

# Abstrakt

Tato diplomová práce se zabývá vývojem hry na mobilní zařízení s operačními systémy iOS a Android OS nazvanou Re!Sources. Tato hra byla vyvíjena společností Zoongo, s.r.o. V textu je popsán celý vývojový cyklus hry, hra samotná a její implementace. Hra byla vyvíjena v nástroji pro vývoj multiplatformních aplikací Shiva 3D společnosti Stonetrip. Programovacím jazykem byla Lua, jež je standardním jazykem nástroje Shiva 3D.

# Klíčová slova

Shiva 3D, Lua, iOS, Android OS, iPhone, iPad, hra, mobilní zařízení, Re!Sources

# Abstract

This thesis deals with the development of games on mobile devices with operating systems iOS and Android OS called Re!Sources. This game was developed by Zoongo s.r.o. In the text part of thesis is described the entire development cycle of the game, the game itself and its implementation. The game was developed in the tool for the development of multiplatform applications Shiva 3D from Stonetrip. Programming language was Lua, which is the standard language for Shiva 3D.

# Key words

Shiva 3D, Lua, iOS, Android OS, iPhone, iPad, game, mobile device, Re!Sources

# Použité zkratky

| | |
|---|---|
| SDK | Software Development Kit |
| WYSIWYG | What You See Is What Yout Get |
| DWF | Design Web Format |
| VoIP | Voice Over Internet Protocol |
| SVN | Subversion |
| OOP | Objektově orientované programování |
| OO | Objektově orientovaný |
| UML | Unified Modeling Language |
| HUD | Head-Up Display |
| MMO | Massive(ly)-Multiplayer Online |
| GPS | Global Positioning System |
| XML | Extensible Markup Language |
| IT | Information technology |

# Obsah

# 1 Úvod

Mým tématem diplomové práce byl vývoj herní aplikace na platformu iOS a Android OS. Na předmětu své práce jsem pracoval v rámci startupu společnosti Zoongo, s.r.o. zaměřené na vývoj smart phone a tablet aplikací. V této společnosti jsem pracoval v týmu pěti lidí, kde každý měl na starost pouze určitou část vývoje. S tímto týmem jsme úspěšně vydali a umístily hru na Apple App Store. Mým úkolem byl návrh a implementace herní logiky, práce s 2D objekty hry a jejich interakce s 3D prostředím.

Na začátku se dozvíme něco o implementačním prostředí, jež jsme používali. Je zde popsán nástroj, který jsme požívali, jeho srovnání s konkurencí a popis jazyka, ve kterém jsme programovali.

Další kapitola popisuje vývojový cyklus. Jsou zde odlišnosti vývojového cyklu hry od vývojového cyklu jakéhokoliv jiného komerčního softwaru. A je zde také popsán celý životní cyklus naší hry nazvané Re!Sources od specifikace záměru až po nasazení.

Poslední kapitola je zaměřena na jednotlivé části hry, na kterých jsem pracoval. U každé části je popsán její návrh i samotná implementace.

# 2 Implementační prostředí

Problematika multi-platformního vývoje v generickém prostředí jednotlivých platforem a nutnost vyvíjení, testování a udržování jedné aplikace v různých vývojových prostředích (v javě pro Android OS, v objective-C pro iOS, apod.) generuje:

- násobné finanční náklady pro vývoj a testováni aplikací,
- zvýšené riziko výskytu chyb a multiplikace testovacích scénářů,
- zvýšené nároky na pracovníky vývojových a testovacích týmů.

Proto jsme zvolili nástroj Shiva 3D [7], který umožňuje vývoj držet na jedné platformě, kromě samotného nasazení.

## 2.1 Shiva 3D

Shiva 3D je vývojářský nástroj pro snadné vytváření 3D real-time aplikací a her pro Windows, Mac OS, Linux, iPhone, Android, BlackBerry, Palm, Wii a iPad a to jako samostatné aplikace, nebo vložené do webových prohlížečů.

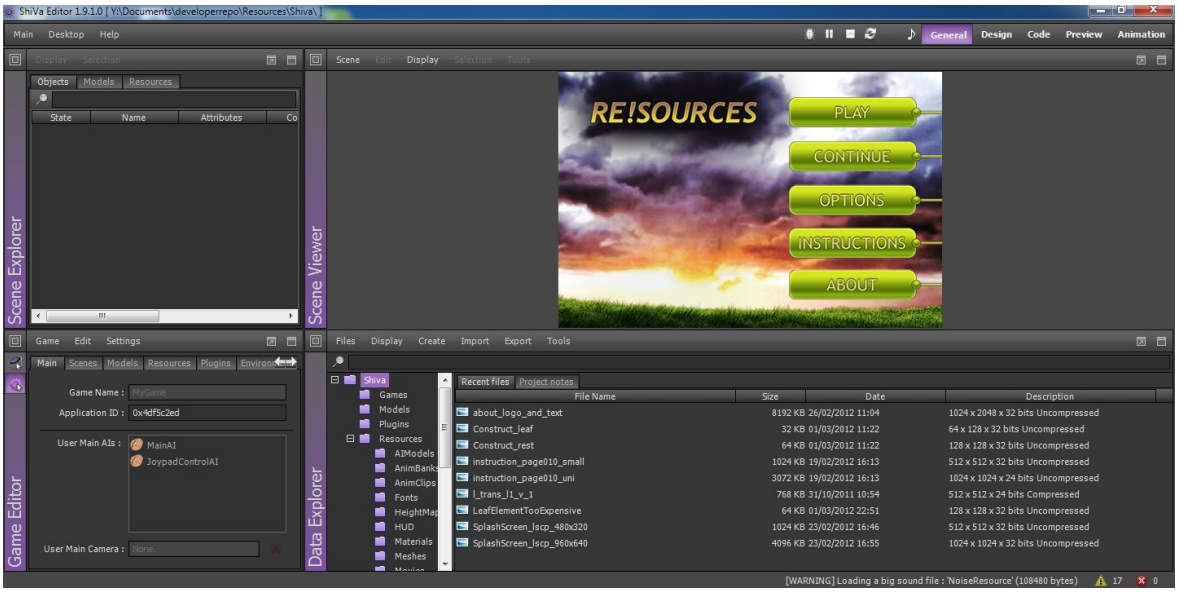
Herní engine využívá OpenGL, OpenGL ES a DirectX, a může být také spuštěn v režimu software. ShiVa 3D také podporuje standardní plug-iny, jako je NVIDIA PhysX , F-Mod zvukové knihovny a ARToolKit.

Shiva 3D se zkládá z:

- Shiva 3D Editor
- Shiva 3D Authoring Tool
- Shiva 3D Server

### 2.1.1 Shiva 3D Editor

Editor je WYSIWYG software pro navrhování her a aplikací. Je to stěžejní nástroj, ve kterém se „vyrábí“ celá aplikace. Dají se zde importovat věci z 3ds Max, Maya a XSI, textury, zvuky a videa, konfigurovat materiály, částice, terén, oceán a mnoho dalšího. Jak tento editor vypadá lze vidět na obrázku Obr. 1.




*Obr. 1 – Shiva 3D Editor*

Je zde, ale jedna velká nevýhoda a to ta, že Editor je k dispozici pouze pro Windows, což je problém pokud se vyvíjená aplikace distribuuje na iOS, jelikož tyto distribuce jdou dělat jenom na Mac OS. Díky tomu si musí vývojář pořídit Windows, pokud je již nemá.

#### 2.1.1.1 ShiVa Editor Moduly

Shiva, jako stejnojmenný hinduistický bůh, je tvarovatelná. Transformuje výtvor (dílo) do 3D aplikace v reálném čase pomocí jeho mnoha paží, známých jako moduly.

K dispozici je modul pro každou z hlavních funkcí, 21 celkem, které umožňují vytvoření vlastního inovativního rozhraní výběrem pouze modulů, které se potřebují pro konkrétní úkol.

**AI Model Editor**

Umožňuje deklarovat proměnné, funkce, stavy a handlery. AI může být bráno jako třída z objektově orientovaného programování. AI Model Editor pracuje se Script Editorem, který je popsán níže.

**Script Editor**

Tento editor je nejvíce používaný ze všech. Implementuje se zde veškerá inteligence a chování aplikace. Script editor jak už bylo řečeno, pracuje s AI Model Editorem. Scriptování je založeno na optimalizované verzi jazyku Lua [9], programovat lze také přímo v C, C + + a Objective-C podle uživatelova oblíbeného vývojového nástroje.

Script editor zahrnuje:

- Automatický návrh a automatické dokončování.
- Zvýraznění syntaxe.

- Skládání kódu.

- Integrovaná nápověda (F1 na funkci).

Od verze 1.9.1 je přidán i dlouho očekávaný debugger, díky kterému se snáze detekují jednotlivé chyby.

**Data Explorer**
Pro import dat určitě použijete Data Explorer. Lze zde importovat 3D objekty ve formátu Collada nebo DWF, které jsou podporovány většinou programů pro tvorbu 3D. Dále lze importovat většinu zvuků, videí a obrázků a v neposlední řadě také Shiva 3D archívy. Kromě importu nabízí také možnost pro vytváření jednoduchých 3D objektů jako je koule, krychle, válec aj.

**HUD Editor**
Používá se k vytvoření 2D rozhraní, definují se zde obrázky, texty, tlačítka, posuvníky, atd.

**Scene Viewer**
Patří do rodiny WYSIWYG editorů. Slouží k zobrazení jednotlivých 3D modelů, HUD objektů a scény aplikace, tak jak bude vypadat po vyrenderování.

#### 2.1.1.2 Verze Shiva 3D Editoru
Shiva 3D Editoru je k dispozici v několika verzích, které se od sebe liší hlavně po finanční stránce.

- Web - Webová edice je neomezená verze pro publikování 3D na webových stránkách.

- Basic - Pro nezávislé vývojáře a malá studia, poskytuje všechny funkce dostupné v bezplatné verzi a umožňuje exportování aplikace komerčně.

- Advanced - Nabízí nástroje potřebné pro vytvoření složitých aplikací, které se obvykle provádí v týmu více lidí. Tyto nástroje jsou určeny jen ke snížení vývojového času. Aplikace z Basic verze nejsou o nic horší či pomalejší než ty vytvořené v Advanced.

- Educational - Jako Advanced v základní ceně, není povolen žádný komerční vývoj.

### 2.1.2 Shiva 3D Authoring Tool
Authoring Tool je nástroj potřebný k publikaci aplikací na různé koncové zařízení. Z STK archivu vygerovaným Shiva 3D Editorem vygeneruje zdrojové soubory a knihovny pro cílovou platformu (Windows, Mac, Linux, Wii, iPhone, iPad, Android, BlackBerry QNX, HP WebOS a Airplay). K tomu je ale občas zapotřebí speciálních SDK jednotlivých platforem.

### 2.1.3 Shiva 3D Server
Spravuje komunikaci více uživatelů mezi sebou u více-uživatelských aplikací. Lze s ním vytvořit MMO aplikaci. Nejnovější verze obsahuje VoIP, což umožňuje uživatelům mluvit spolu tak moc, jak potřebují.

Shiva Server je k dispozici pro Windows a Linux (FreeBSD, Ubuntu).

### 2.1.4 Porovnání s ostatními nástroji

Shiva 3D není jediný nástroj pro vytváření multi-platformních 3D aplikací. Existuje jich celá řada. Hlavní kritérium při našem hledání výsledného nástroje, ve kterém naši hru zpracujeme, byla podpora distribuce výsledné aplikace na platformy iOS a Android OS. Dalším důležitým faktorem bylo použití 3D. V tabulce Tab. 1, lze vidět porovnání tří různých nástrojů, ze kterých jsme nakonec vybrali Shiva 3D. Nástroje jsou Unity 3D [10], Unreal Engine 3 [11] a Shiva 3D.

| |  **Unity 3D** |  **Unreal Engine 3** |  **Shiva 3D** |
|---|---|---|---|
| **Podporované platformy** | Webové prohlížeče,<br>Adobe Flash,<br>iOS,<br>Android,<br>Windows,<br>Mac OS,<br>Nintendo Wii,<br>PlayStation 3,<br>Xbox 360 | Adobe Flash,<br>iOS,<br>Android,<br>Windows,<br>Mac OS,<br>Xbox 360,<br>PlayStation 3,<br>PlayStation Vita | Windows,<br>Mac,<br>Linux,<br>iOS,<br>Android,<br>Palm,<br>Nintendo Wii,<br>webové prohlížeče |
| **Vývojové prostředí** | Unity Editor | Unreal Editor | Shiva 3D Editor |
| **Programovací jazyk** | C#,<br>JavaScript,<br>Boo | UnrealScript | Lua,<br>C,<br>C++,<br>Objective-C |
| **Cena** | cca 1000 $ | 1000 $ | 400 $ |

*Tab. 1 – Porovnání nástrojů*

## 2.2 Lua

Lua je přenositelný, silný, rychlý, lehký, zabudovatelný skriptovací jazyk.

Její silnou stránkou je práce s datovými strukturami, podpora asociativních polí a rozšiřitelné sémantice. Lua je dynamicky typována, běží na interpretovaném bytekódu pro virtuální stroje založených na registrech a obsahuje automatickou správu paměti s postupným uvolňováním paměti, díky čemuž je ideální pro konfiguraci, scriptování a rychlé prototypování.

Lua je navržena, implementována a udržována týmem na PUC-Rio, Pontifical Catholic University of Rio de Janeiro v Brazílii. Lua byla zrozena a vyvíjena v Tecgraf, the Computer Graphics Technology Group of PUC-Rio a nyní se nachází v Lablua. Obě laboratoře Lablua a Tecgraf jsou laboratoře katedry informatiky PUC-Rio.

### 2.2.1 Výhody

- *Lua je robustní a osvědčený jazyk:*

Lua byla použita v mnoha průmyslových aplikacích (např. Adobe Photoshop Lightroom), v aplikacích s důrazem na zabudované systémy (např. Ginda middleware pro digitální televizi v Brazílii) a hry (např. World of Warcraft). Lua je v současné době předním skriptovacím jazykem ve hrách. Lua má solidní referenční manuál s několika knihami o něm. Několik verzí Lua byly vydány a používány v reálných aplikacích od svého vzniku v roce 1993. Aktuální verze Lua je 5.2 vydána 6 prosince 2011.

- *Lua je rychlá:*

Lua má zaslouženou reputaci za výkon. Tvrzení „být rychlý jako Lua“ je touha dalších skriptovacích jazyků. Několik srovnávacích testů ukazují jazyk Lua jako nejrychlejší jazyk v oblasti interpretovaných jazyků.

- *Lua je přenosná:*

Lua je distribuována v malém balení a vychází out-of-the-box na všech platformách, které mají ANSI/ISO C kompilátor. Lua běží na všech zařízeních Unixu a Windows, ale i na mobilních zařízeních (jako jsou kapesní počítače a mobilní telefony, které používají BREW, Simbian, Android, atd.) a vestavěných mikroprocesorů (jako je ARM a Rabbit) pro aplikace jako je Lego MindStorms.

- *Lua je zabudovatelná:*

Lua je rychlý jazykový engine s malou velikostí, který lze snadno vložit do aplikace. Lua má jednoduché a dobře zdokumentované API, které umožňuje silnou integraci s kódem napsaných v jiných jazycích. Je snadné rozšířit jazyk Lua s knihovnami napsanými v jiných jazycích. Je také snadné rozšířit programy napsané v jiných jazycích s jazykem Lua. Lua byla použita k rozšíření programů napsaných nejen v C a C++, ale také v jazyce Java, C#, Smalltakl, Fortran, Ada, Erlang, a to i v jiných skriptovacích jazycích jako je Perl a Ruby.

- *Lua je výkonná (ale jednoduchá):*

Základním konceptem návrhu jazyku Lua je poskytovat meta-mechanismy pro implementování vlastností, místo poskytování mnoha vlastností přímo v jazyce. Například, ačkoliv Lua není čistě objektově orientovaný jazyk, poskytuje meta-mechanismy k implementaci tříd a dědičnosti.

- *Lua je malá:*

Přidání Lua do aplikace neznamená její zvetšení. Archiv Lua 5.2.0, který obsahuje zdrojové kódy a dokumentaci zabírá 241 KB v komprimované podobě a 950 KB v nekomprimované. Zdroj obsahuje kolem 20 tis. řádků v C.

- *Lua je zdarma:*

Lua je bezplatný open-source software šířený pod velmi liberální licencí (velmi známá MIT licence). Může být použita pro jakýkoliv účel, včetně obchodních účelů a to bez jakéhokoliv poplatku. Stačí stáhnout a používat ji.

### 2.2.2 Nevýhody

- *Lua je malá:*

Tím, že je malá, neosahuje velké knihovny velmi užitečnými funkcemi, které obsahují jiné programovací jazyky.

- *Lua není objektově orientovaná:*

Lua není objektově orientovaný jazyk. Pokud bychom chtěli používat OOP, musíme si vytvořit svůj vlastní OO systém za použití meta-metod a přizpůsobit si ho vlastním potřebám.

- *Lua používá jeden typ pro čísla:*

Lua nepodporuje 2 typy čísel, jeden pro celočíselné hodnoty a druhý pro hodnoty s desetinou čárkou, což může vést k ohromujícímu zpracování.

- *Lua není standardizována:*

Díky jejímu nevelkému rozšíření není zavedena jako “de fakto standard”.

# 3 Struktura vývojového cyklu

V současné době většina vývojových týmu zaměřených na vývoj komerčně používaného softwaru dělí vývojový projekt do fází kopírujících standardní vývojové a implementační metodiky. Obvyklé dělení sleduje přibližně tuto strukturu:

1. Sběr požadavků:
    a. funkční a nefunkční požadavky,
    b. formulace business zadání.
2. Případy užití a logické vymezení systému.
3. Detailní logický model:
    a. business třídy,
    b. klíčové přechodové stavy,
    c. popis interakce jednotlivých tříd.
4. Fyzický návrh:
    a. návrh fyzických tříd a jejich atributů a metod,
    b. návrh fyzického datového modelu,
    c. návrh uživatelského rozhraní.
5. Návrh nasazení aplikace na jednotlivé komponenty IT infrastruktury

Vývojový cyklus hry je v mnoha ohledech společný s vývojovým cyklem jakéhokoliv jiného komerčního softwaru. Odlišnosti spočívají dle mého pohledu zejména v následujících oblastech:

1. Použiti grafického a multimediálního obsahu, vynucující balanci mezi výkonem zařízení a kvalitou grafického, video, respektive audio obsahu.

2. Rozmanitost cílových zařízení, pro než je vyvíjený software určen a to nejen z pohledu operačního systému, ale i z pohledu zobrazovacích možnosti jednotlivých zařízení, jejich ovládaní a výbavy dodatečnými funkcemi (pohybový senzor, GPS, apod.)

3. Rozšířeni vývojového cyklu, vidím v tom, že součásti sestavení verze a nasazení kroku je integrace s validačními a schvalovacími procedurami distribučních a prodejních platforem (App Store, Play).

4. Vývoj většiny komerčního softwaru má analytik a návrhář přímý kontakt s budoucím uživatelem, respektive business garantem. V případě návrhu hry je vzhledem k velikosti trhu tento kontakt omezený a reálný budoucí uživatel je často nahrazován tržními a marketingovými analýzami.

5. Dalším rozšířením oproti standardním aplikacím je nutnost návrhu zvukové složky hry a to nejen ve vazbě na celkové zvukové pozadí hry, ale i na zvukové a ruchové charaktery jednotlivých objektů.

V rámci jednotlivých přírůstků docházelo k rozšiřování, jednak vlastní herní logiky, detailu herních objektu a vylepšování textur. Souběžně s jednotlivými přírůstky projektu docházelo k ladění parametrických dat, které mají vliv na obtížnost a hratelnost hry a k hernímu balancováni jednotlivých úrovní.

Pro komunikaci mezi členy týmu bylo využíváno vybraných typů UML diagramů. Zejména byly využívány tyto diagramy:

- *Diagram tříd*
- *Diagram aktivit*
- *Stavový diagram*

## 3.1 Specifikace záměru

Dnes již existuje nepřeberné množství her, jež se dělí do různých skupin, podle toho co je účelem hry. Těmto skupinám se v herní terminologii říká „herní žánry“ [5]. Podle těchto žánrů si pak hráč může vybrat hru, která ho bude bavit.

*Adventura* – tyto hry mívají propracovaný příběh, kde hráč v roli hlavního hrdiny musí řešit hádanky s různou obtížností, hledat, sbírat a používat předměty, jež mu pomáhají při putování světem a splnění hlavního úkolu. Při putování světem se setkává s obyvateli tohoto světa, se kterými může hráč vést dialog (rozhovor), který mu může pomoct při řešení nějakého úkolu nebo mu může nějaký úkol zadat.

*Akční hra* – v těchto hrách je hráčova hlavní náplň eliminace nepřátel. Tito nepřátelé disponují určitou umělou inteligencí, aby byli schopni se bránit. Na hráče jsou zde kladeny nároky na rychlé reflexy čí precizní míření. Většina těchto her obsahuje možnost hraní proti opravdovým hráčům místo hraní proti umělé inteligenci.

*Arkáda* – pod tímto žánrem se skrývají hry, jež jsou většinou rozděleny na více kol, u kterých se stupňuje obtížnost. Tyto kola mohou být také časově omezena. Také zde bývají upraveny fyzikální zákony. Od hráče se zde očekává zejména postřeh, rychlost logického myšlení a předvídání.

*Simulátor* – V tomto žánru se skrývají hry, jež se snaží co nejvíce přiblížit reálnému světu. Ať se jedná o nejrealističtější grafické znázornění světa nebo precizně vypracovány fyzikální zákony.

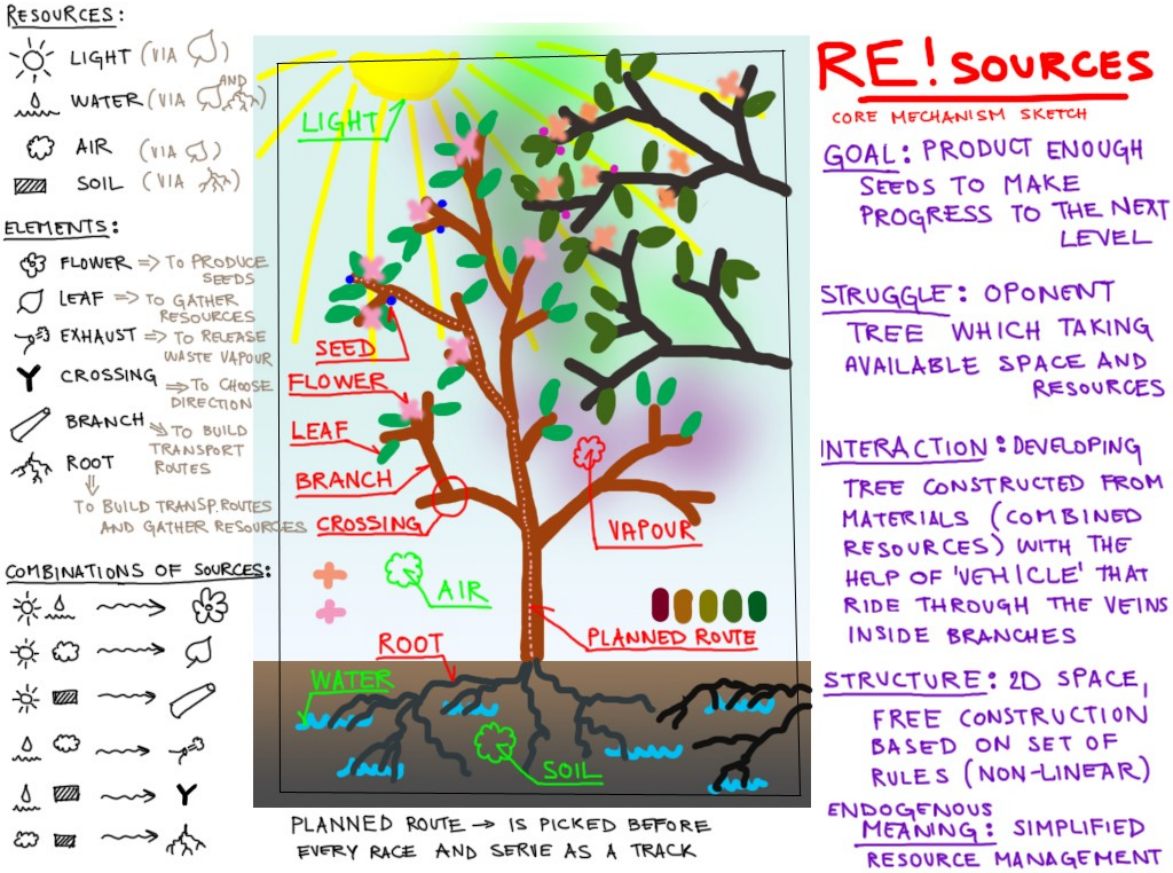
*Strategie* – u her, které spadají pod tento žánr, je hlavní důraz kladen na strategické myšlení hráče. Hráč je zde často v roli nějakého velitele, který ovládá větší skupinu objektů (lidi, budovy, stroje, aj.) a přiřazuje jim různé úkoly, které vedou ke zničení nepřátel. Pro vytváření těchto objektů nebo jejich vylepšování je zapotřebí dostatek surovin. Jako u akčních her, tak i většina strategií obsahuje možnost hraní proti opravdovým hráčům místo hraní proti umělé inteligenci

Jelikož existuje celá řada her, všech možných žánrů, snaží se vývojáři přicházet s jedinečnými hrami, jež by hráče zaujaly svou originalitou. K tomu často kombinují různé herní žánry

dohromady. Naše hra není výjimkou. Abychom měli nějakou šanci na úspěch, museli jsme přijít s takovou hrou, která se bude od ostatních lišit a zároveň bude spadat do masově oblíbených žánrů. A tak se zrodil nápad vytvoření více úrovňové arkády prvky real-timové strategie, díky oblíbenosti spojení těchto dvou žánrů. S tímhle požadavkem jsme zároveň splnily náš cíl ověřit života schopnost konceptu hry s dynamicky proměnou strategií závislou na chování hráče. Tato hra nese název Re!Sources.

## 3.2 Herní koncept

V rámci herního konceptu jsme na opakovaných workshopech upřesňovali algoritmicky i graficky koncept hry a upřesňovali specifikace scénářů, typu výhra, prohra. Jak vypadal původní návrh hry, se můžete podívat níže na obrázek Obr 2.




*Obr. 2 – Původní návrh*

Ve finálním řešení jsme upustily od nutnosti kombinace různých zdrojů pro stavění jednotlivých částí stromu, jež jsme nahradily celkovou energií. Také jsme upustily od možnosti stavění výfuků a udělali mnoho dalších úprav.

Na začátku hry začíná hráč s malým stromem, který v průběhu hry rozšiřuje. Má na výběr ze dvou pohledů. První je pohled mapy, který slouží ke stavění stromu a pozorování okolního světa. Druhý pohled je uvnitř stromu a zde hráč ovládá vážku, se kterou sbírá natěžené zdroje. Tyto zdroje se přeměňují na energii, která je nutná ke stavění a údržbě stromu.

Pokud hráč na stromě postaví určený počet plodů, vyhrává tuto úroveň. Pokud vyhraje všech 10 úrovní, které hra obsahuje, vyhrává celou hru. Pokud se ale v průběhu hry dostane s energií pod určitou hranici, prohrává tuto úroveň a musí si ji zopakovat.

## 3.3 Detailní návrh hry

Detailní návrh byl v našem vývojovém cyklu specifikován dokumentem detailní návrh. Ten jasně formuloval přesný popis klíčových algoritmů potřebných pro hru jako počítání energie, budováni stromu, určeni pozice hráče uvnitř stromu, energetická bilance stromu a bilance vitality.

Jak už bylo řečeno, hlavním úkolem hry je vypěstovat určitý počet plodů na svém stromě, což hráčovi umožní postup do dalšího úrovně. Snažíme se zde simulovat reálný svět, ve kterém růst stromu ovlivňuje mnoho faktorů. Jako je získávání vody, živin z půdy, dále jen půda, světla a vzduchu dále jen zdrojů a různých překážek, skal a větví sousedních stromů.

Abychom se přiblížili skutečnému stromu, má hráč možnost postavit listy, květy, větve a kořeny. Dále se na stromě vyskytují plody, které sice nejdou přímo postavit, ale vznikají z květů.

**List** – list slouží k získávání světla.

**Květ** - květ nezískává žádný zdroj, přesto je ale velmi důležitý. Po nějakém čase se totiž změní na plod.

**Plod** – plody také nic netěží, ale určitý počet je vždy nutná podmínka pro vyhrání úrovně.

**Větev** – na výběr je ze tří druhů větví, které na sebe vzájemně navazují. Umisťují se nad zemský povrch a lze na ně stavět listy a květy. Každá větev získává vzduch, a pokud je na ni umístěn ještě i list, tak získává i světlo.

**Kořen** – na výběr je také ze tří druhů kořenů vzájemně na sebe navazujících, jako je tomu u větví. Kořeny se, jak už název napovídá, umísťují pod zemský povrch a slouží k získávání vody a půdy. Ty získávají, pokud narazí na nějaké ložisko vody nebo půdy. Na kořeny samozřejmě nelze stavět listy ani květy.

Zdroje, které strom získává z okolního světa, se objevují uvnitř stromu ve formě barevných kuliček a po jejich sebrání se přemění na energii, kterou určitým počtem navýší. Každý list, květ, plod, větev i kořen spotřebovávají nějakou energii a proto je potřeba ji neustále doplňovat. Energie neslouží jenom k údržbě stromu, ale také ke stavění nových částí stromu.

Hlavní postavou hry je speciální tvor žijící uvnitř stromu, který mu pomáhá k životu a růstu a podobá se vážce z reálného světa. Hráč se tímto tvorem pohybuje po celém stromě a sbírá v něm zdroje pro další růst a udržení stromu při životě.

Hra se skládá ze dvou pohledů. Prvním z nich je mapa. Ta je celá ve 2D a slouží ke stavění a pozorování okolního světa. Druhý pohled, který hra nabízí je pohled uvnitř stromu. Ten je už ve 3D a hráč ho použije k pohybu uvnitř stromu a sbírání natěžených zdrojů. V obou pohledech jde vidět celkové energie stromu a zbývající počet plodů na výhru úrovně. Podrobněji se na oba pohledy podíváme dále.

Hra má celkově 10 úrovní, po jejichž úspěšném ukončení hráč zvítězí. Každá úroveň má určený počet plodů pro postup do další úrovně. Pokud se ale hráči podaří během hry dostat s energií pod určitou hranici, která se v jednotlivých úrovních může lišit, je hráč nucen si tuto úroveň zopakovat od začátku.

### 3.3.1 Pohled mapa

Jak už bylo řečeno výše, mapa je celá ve 2D a slouží ke stavění a pozorování okolního světa. V tomto pohledu vidí hráč svůj strom, okolní stromy a zdroje v půdě. Tento pohled je zobrazen na obrázku Obr. 3.


*Obr. 3 – Pohled mapa*

U svého stromu může vidět, kde má postavený list, kolik květů se ještě musí přeměnit na plody, které kořeny se dotýkají zdrojů v půdě, atd. Také další velice důležitá informace, kterou hráč může vidět na svém stromě je poloha vážky sloužící k orientaci v pohledu uvnitř stromu. K orientaci uvnitř stromu poslouží také vlaječka, jež se umisťuje na mapě.

U konkurenčních stromů bude hráče nejvíce zajímat, jak daleko mají postaveny svoje větve a jestli už ho náhodou neobklíčili.

S mapou se dá jednoduše pohybovat do všech směrů jednoduchým tažením prstu a také se dá přibližovat a oddalovat jednoduchými gesty dvou prstů.

Na mapě je vidět celková energie stromu a zbývající počet plodů na výhru úrovně. Mimo tyto ukazatele obsahuje mapa čtyři tlačítka. První z nich je tlačítko pro přepnutí do menu, které slouží také jako pauza. Nachází se nahoře uprostřed.

Další tlačítko nacházející se vlevo dole, slouží ke stavění. Po jeho zmáčknutí se otevře stavěcí menu, které nabízí postavení třech druhů větví/kořenů, listu a květu. K tomu aby hráč mohl svůj strom rozšiřovat, potřebuje dostatek energie a volné místo k růstu. Pokud jsou obě tyto podmínky splněny, může se strom rozrůst. Ale pozor nejde stavět cokoliv a kdekoliv, stavět se dá jenom na té větvi nebo kořenu, ve kterém se hráč právě nachází s vážkou uvnitř stromu. V tomto menu je ukázáno, jaké části stromu se dají postavit a které ne. Ty, které se postavit nedají kvůli nedostatku místa nebo volné pozice jsou zbarveny červeně. A jelikož každý list, květ, větev nebo kořen stojí nějakou energii, může se stát, že na nějaký z nich nebude dostatek energie, ty jsou zbarveny bíle a je přes ně červený znak dolaru. Jakmile si hráč vybral, co chce postavit a splnil všechny podmínky k tomu, stačí ji jednoduchým kliknutím a přetáhnutím na vybrané místo postavit. Po kliknutí se objeví na mapě bíle zbarvené volné pozice, kde se dá objekt umístit. Při jeho přetahování se některé z nich (vždy maximálně jenom jedna) zbarví červeně. Po odkliknutí se vybraná část stromu postaví na tu pozici, na které je aktuálně zbarvená do červena. Pokud se hráč rozmyslí a vybraný element nechce nikde postavit, stačí ho přetáhnout tak aby žádná z volných pozic nebyla zbarvena do červena a odkliknout.

Spodním tlačítkem uprostřed se umisťuje vlaječka, k navigaci uvnitř stromu. A posledním spodním tlačítkem se přepneme do pohledu uvnitř stromu.

### 3.3.2 Pohled uvnitř stromu

V tomto pohledu setrvá hráč většinu herního času. Pohybuje se zde vážkou a pomocí ní sbírá natěžené zdroje. Pohled uvnitř stromu je zobrazen na obrázku Obr. 4.


*Obr. 4 – Pohled uvnitř stromu*

Pro pohyb uvnitř stromu jsou zde 2 joypady a jedno tlačítko. První joypad kruhového tvaru se nachází v levém spodním rohu. Ten slouží k otáčení vážky po obvodu větve/kořene. Na pravé straně se nachází druhý joypad. Ten slouží k posunu vážky dopředu různou rychlostí a má tvar vertikálního posuvníku. Pod tímto joypadem se nachází tlačítko, po jehož zmáčknutí se vážka otočí o 180 stupňů.

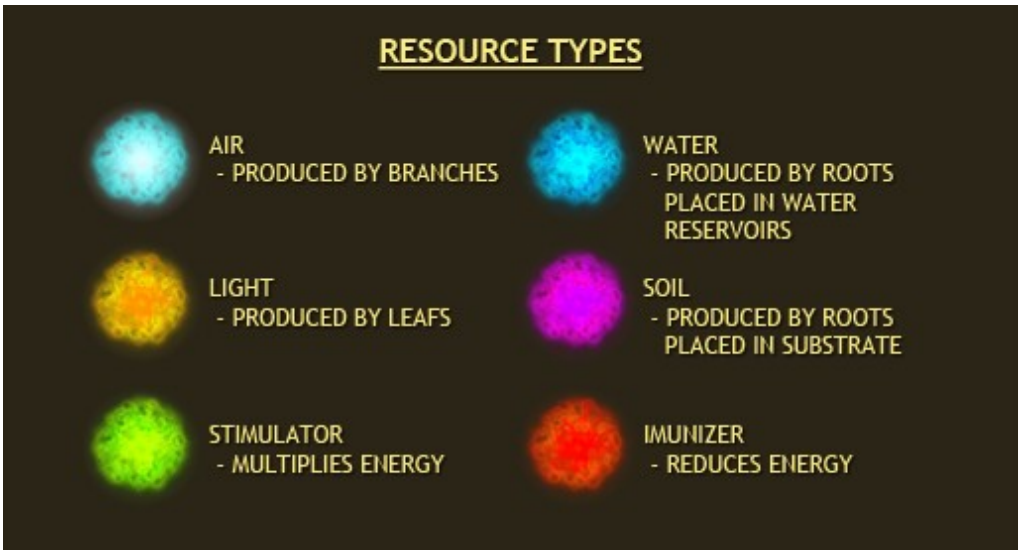
Tak jako jsou na mapě v horní části displeje dva indikátory a tlačítko na přepnutí do menu, jsou tyto objekty v tomto pohledu také. Ještě je zde poslední tlačítko, jež je umístěno nad levým joypadem, sloužící pro přepnutí do mapy.

Pokud si hráč na mapě postavil vlaječku na nějakou větev / kořen, tak se mu uvnitř stromu vytvoří šipky. Tyto šipky jsou na všech křižovatkách a navádí hráče až do cílové větve/kořene. Aby hráč věděl, že se dostal do svoji vybrané větve nebo kořene, je zde zobrazena vlaječka totožná s tou, kterou si hráč položil na mapě.

### 3.3.3 Zdroje

Zdroje, které strom získává z okolního světa, se zobrazují ve formě barevných kuliček a po jejich sebrání se přemění na energii, kterou určitým počtem navýší. Vzduch, jehož je v celém stromu

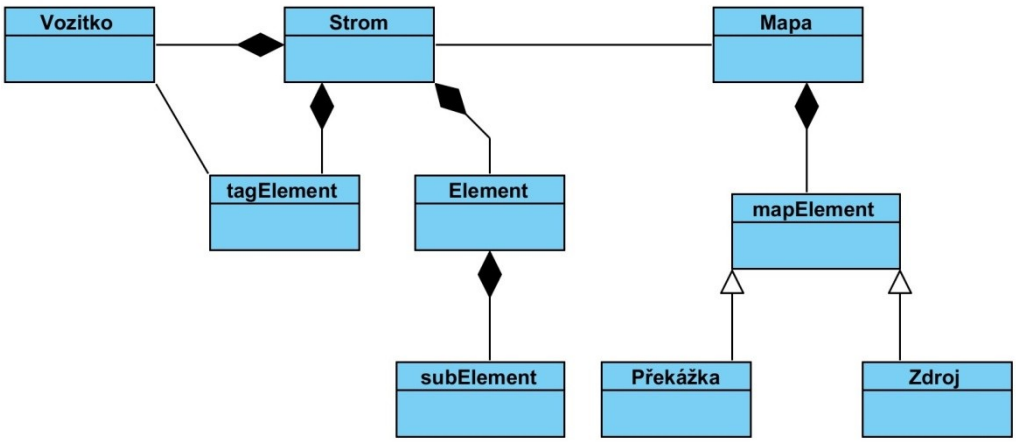
nejvíce, zvýší energii jenom nepatrně. Světlo, kterého už je méně, ale přece jenom dost, pokud má strom dost listů, zvýší energii docela znatelně. Půda už je vzácnější a její zdroje jsou limitované, proto přidá energie velké množství. A pak je tu voda, ta je také vzácná jako půda a její zdroje jsou taky limitovány, ale ta hráčovi přidá extrémní množství energie. Kromě již zmiňovaných čtyř zdrojů se ve stromě objevují další dva. Jeden, který energii výrazně navýší, ale jenž se objevuje málokdy a druhý, který jí zase sníží a objevuje se hodně často. Typy zdrojů jsou vyobrazeny na obrázku Obr. 5.




*Obr. 5 - Zdroje*

## 3.4 Logicky návrh aplikace

Tato fáze projektu byla do značné míry strukturovaná kroky herního konceptu a detailního návrhu hry. Z pohledu „klasického“ životní cyklu návrhu aplikace jsme zde použily jediný výstup, logický diagram tříd zobrazený na obrázku obr. 6.




*Obr. 6 – Logický diagram tříd*

## 3.5 Fyzicky návrh

Fyzicky návrh byl determinován těmito klíčovými faktory:

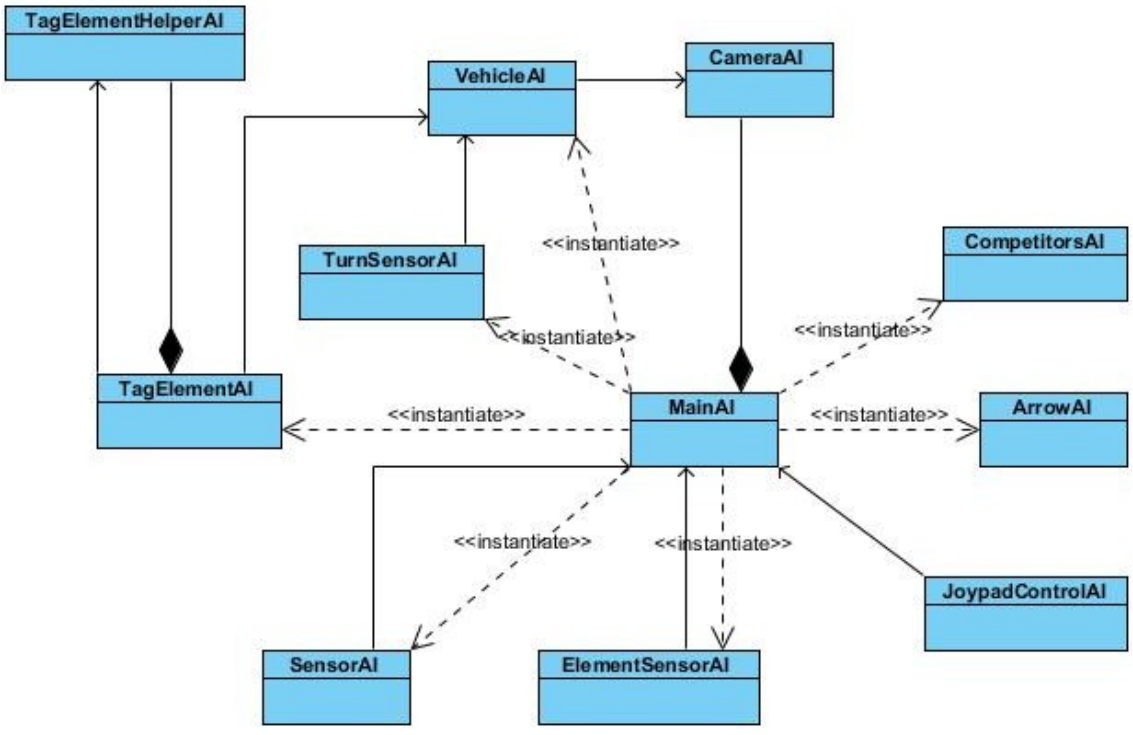
1. Cílovým prostředím – smartphony a mobilní telefony.
2. Multi-platformním vývojovým frameworkem Shiva 3D.

Zvláštnosti a omezení návrhu aplikací pro smartphony a mobilní telefony:

- Na rozdíl od dnes již standardních vícevrstvých architektur, které umožňují striktní oddělení dat od business logiky a funkčnosti vázaných na formulář, respektive displej aplikace, Shiva 3D nemá implementován žádný systém relační databáze. A práce s daty je omezena pouze na interní tabulky a XML soubory. Práce s formulářem je zde nahrazena objektem HUD.

- Design aplikace jsme museli přizpůsobit specifickému prostředí aplikace. Tímto prostředím byl multi-platformní vývojový framework Shiva 3D společnosti Stonetrip.

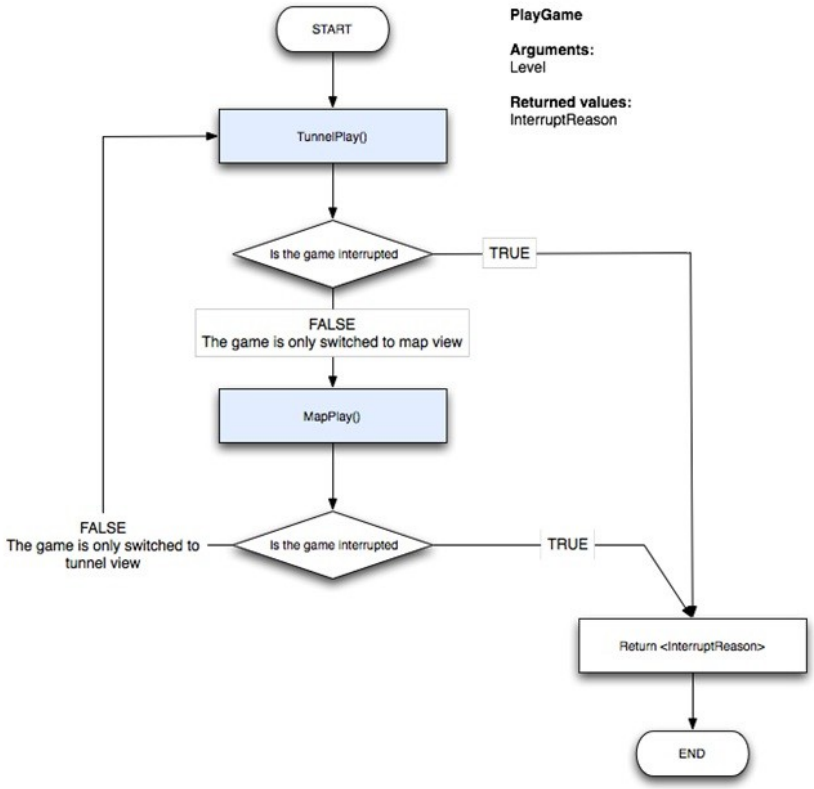
Na obrázku obr. 7, je k vidění diagram fyzických tříd. Lze z něj vyčíst, že hlavní třída MainAI vytváří instance všech ostatních tříd. Jedinou instanci třídy, kterou MainAI nevytváří je JoypadControlAI. Tu i s MainAI vytváří Shiva 3D při spuštění aplikace.




*Obr. 7 – Diagram fyzických tříd*

### 3.5.1 Funkční návrh

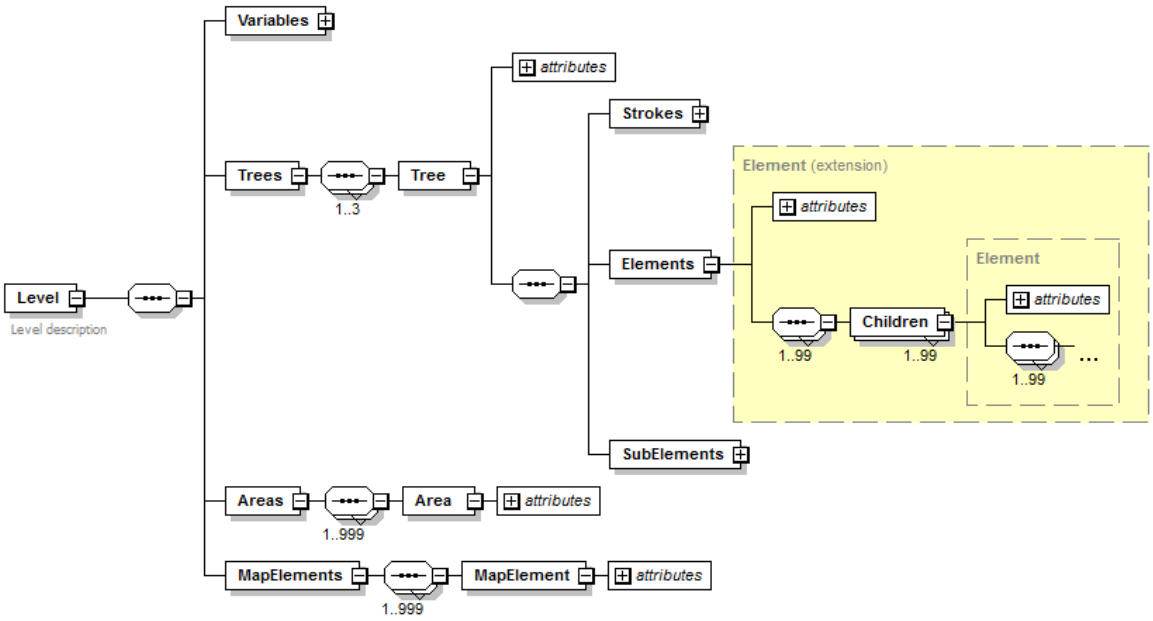
Pro funkční návrh jsme používali aktivity diagramy. Tyto diagramy používaly určitou míru abstraktnosti, nebyly zde názvy. Byly použity nejen pro funkční návrh, ale i pro funkční návrh metod. Podle těchto diagramů se poté programovalo. Na obrázku Obr. 8, je zobrazen jeden z těchto diagramů. Tento diagram popisuje hraní úrovně, kdy se hráč může přepínat mezi dvěma pohledy.




*Obr. 8 – Diagram PlayGame*

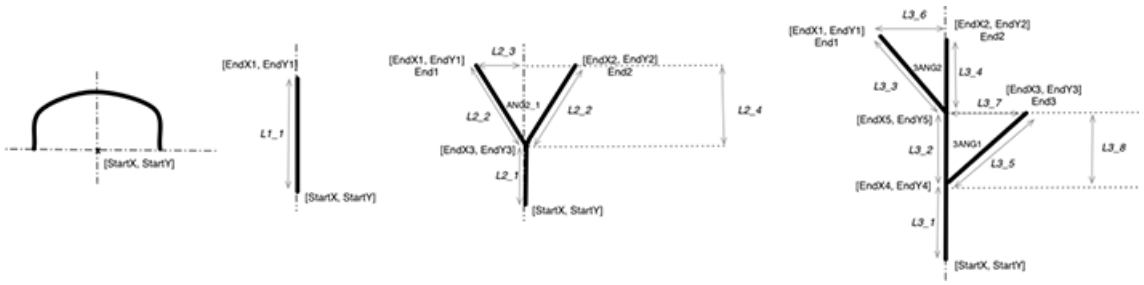
### 3.5.2 Datový návrh

Vzhledem k tomu, že se v průběhu implementace ověřilo jako nejlepší ukládání jednotlivých úrovní do XML, použili jsme XML strukturu zobrazenou na obrázku Obr. 9. K tomu jsme zároveň přizpůsobovali návrh fyzického datového modelu.




*Obr. 9 – XSD struktura XML*

Tuto strukturu obsahuje každá úroveň. Kořenový element *Level* obsahuje konkrétní informace pro jedenu úroveň. V elementu *Variables* jsou uloženy proměnné, které specifikují danou úroveň. Je to nejen jedinečný identifikátor úrovně, počet plodů na výhru úrovně, agresivita protihráčů, ale i pozadí úrovně a textury použité uvnitř stromu a mnoho dalších. Následuje seznam stromů s jejich větvemi a subelementy (listy, květy, plody). Každá větev obsahuje atributy, jež ji specifikují. Mezi hlavní atributy patří *IDElement*, který obsahuje jedinečný identifikátor větve. Dalším neméně důležitým je *TypeElement*, jež může nabívat hodnot 0-3, podle toho o jakou větev se jedná. Typem 0 se označuje speciální element, který se na mapě nevykresluje, ale vykresluje se uvnitř stromu v podobě zátky a slouží v něm k uzavření tunelu. Atributem *MasterType* se specifikuje, zdali se jedná o větev nebo kořen. Atribut *Angle* určuje natočení elementu oproti ose rodičovského elementu. Posledními atributy jsou atributy na určování souřadnic jednotlivých elementů, jsou to *StartX* a *StartY* pro určení počátku a *EndX(1-3)* a *EndY(1-3)* pro určení konců elementů. Jednotlivé tvary elementů a jejich startovní a koncové souřadnice lze vidět na obrázku Obr. 10.




*Obr. 10 – Typy elementů se souřadnicemi*

Element *Areas* obsahuje seznam oblastí vody a půdy nebo skal v zemi. Každá oblast je specifikována jedinečným identifikátorem, počátečním obsahem, aktuálním obsahem a kategorií do které patří (voda, půda nebo skála). V elementu *MapElements* je seznam mapových elementů obsahujících atributy *StartX*, *StartY* pro určení souřadnic na mapě, *IDArea* pro určení oblasti, ke které patří a *Category* zdali se jedná o vodu, půdu nebo skálu.

Součástí tohoto designu bylo i detailní 3D objektů a textur. Souběžně s fyzickým návrhem vznikaly první zvukové vzorky.

### 3.5.3 Návrh uživatelského rozhraní

Na rozdíl od klasického vývojového cyklu bylo uživatelské rozhraní navrženo již v předchozích fázích a bylo součástí grafických návrhů. Proto jsem se s ním na úrovní fyzického návrhu nezabýval. Respektive zabýval jsem se pouze vazbou jednotlivých grafických prvků na interakci s funkční logikou.

## 3.6 Vývoj

Vzhledem k organizaci týmu, který byl geograficky distribuovaný mezi okolí Ostravy, Prahu a Gallway (Irsko), bylo nutné použít adekvátní vývojovou infrastrukturu. Ta byla složena z vývojového serveru, na němž běžel systém Apache Subversion neboli SVN [4] a lokálních vývojářských pracovišť, která byla tvořena softwarem společnosti Stonetrip. Vybraná vývojářská pracoviště umožňovala vytváření distribučních balíku. Tato pracoviště byla vybavena operačním systémem OSX a vývojářským balíkem Xcode.

SVN se řadí mezi centralizované verzovací systémy. To znamená, že všechna data jsou uložena na jednom centrálním serveru, který provádí správu verzí. Uživatel si tak musí svá data na lokálním úložišti aktualizovat ze serveru a po provedení změn je zpátky na server nahrát. Při nahrávání je dobré psát textový popis změn, jež provedl. Díky tomu se dá lépe orientovat v jednotlivých verzích aplikace. SVN nám také umožňuje vrátit se k verzi z minulosti, která byla aktuální například před měsícem. Při nahrávání jakékoliv změny se zvýší tzv. číslo revize a na server se nenahraje celý soubor, ale jen to, k jakým změnám došlo oproti poslední revizi.

## 3.7 Testováni

V rámci testování probíhali testy jednotlivých verzí hry s důrazem na stabilitu, na korektnost zobrazování grafiky a plynulý běh hry. Dělali se unit testy, celkový regresní test aplikace a uživatelské testy. Testovala se nejen hra, ale i editor úrovní a jednotlivé úrovně, přičemž testování jednotlivých úrovní nebylo zaměřeno jako u aplikace na stabilitu a výkonnost, ale bylo zaměřeno na hratelnost.

Pro interní testování byla použita Bugzilla na kontrolu a hlášení chyb. Pro uživatelské testy byla vytvořena internetová aplikace, která zajišťovala komunikaci s uživatelskými testery.

Jako programátor jsem dělal jednotlivé unit testy. Testování jsem prováděl na iPhone 3G.

## 3.8 Nasazení

Problematika nasazení řeší situaci, kdy hra je implementována v jazyce Lua, který je platformně nezávislý. Cílem je dostat hru do prostředí generického pro danou platformu. V tomto případě pro iOS a Android OS. K tomu Shiva 3D používá specifický postup, popsaný v následující sekci. Tato sekce popisuje postup pro platformu iPhone.

1. *Shiva 3D*
    a. Vygenerování STK balíčku, přes game editor a export.
2. *Shiva Authoring Tool*
    a. Vybrání dané platformy (pro nás iPhone).
    b. Vybrání STK balíčku a dodatečných souborů (videa a xml soubory).
    c. Nastavení ikony a startovního obrázku.
    d. Nastavení parametrů aplikace, jako jsou podporované polohy zařízení, typ distribuce atd.
    e. Vybrání distribučního profilu s nastavením správného identifikátoru.
    f. Vygenerování zdrojových kódů a knihoven pro danou platformu.
3. *Xcode*
    a. Dopřidání ikon a startovních obrázků pro jiná rozlišení.
    b. Nastavení minimální verze systému.
    c. Nastavení správné architektury.
    d. Nastavení jména aplikace.
    e. Zbuildování aplikace pro vytvoření distribučního balíčku.
4. *App Loader*
    a. Nahrání distribučního balíčku do AppStore.
5. *iTunes*
    a. Vyplnění formuláře.
        i. Typ hry,
        ii. možný nevhodný obsah,
        iii. klíčová slova,
        iv. popis.
    b. Nahrání screenshotů ze hry a ikon v několika rozlišeních.

# 4 Samotná implementace

Mým úkolem byl návrh a implementace herní logiky a interakce s 3D světem. To obnášelo modelování objektů ve 2D světě, jejich vzájemnou interakci, vytvoření modelu vitality a práci se zdroji, vytvoření modelu inteligence jednotlivých herních objektů a vytvoření editoru pro návrh jednotlivých úrovní.

## 4.1 Model vitality

Jak už bylo zmíněno výše, hra obsahuje 4 různé zdroje, které zvyšují celkovou energii stromu. Na začátku hra obsahovala 4 indikátory, jeden pro každý zdroj a každá část stromu stála různý počet různých zdrojů. Tento model přinášel další náročnost na pochopení a dělal hru pro hráče komplikovanější. Proto jsme od něj upustili a přešli na jednotnou energii. Tato energie se používá na vše, na stavění i na údržbu stromu. Ke sjednocení všech 4 zdrojů do jednotné energie jsem vyžil vážený průměr [8].

Pro jeho výpočet se potřebují kromě hodnot, u kterých chceme průměr spočítat, také jejich váhy.

Pokud máme dány hodnoty

$$X = \{x_1, \cdots, x_n\},$$

a k nim odpovídající váhy

$$W = \{w_1, \cdots, w_n\},$$

spočítá se vážený průměr tímto vzorcem:

$$\bar{X} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}. \tag{1}$$

Díky tomu se každému zdroji nastaví nějaká váha a podle toho jaká váha se zdrojům nastaví, tak přidají určitý počet energie. Toto nastavení je jedno z mnoha, které určují obtížnost jednotlivých úrovní.

## 4.2 Modelování objektů ve 2D světě

2D světem se zde myslí mapa, která je kompletně ve 2D. Na mapě jsou zobrazeny všechny skály, zdroje vody a živin, stromy s jejich listy, květy i plody. Dále se s mapou dá posouvat a zvětšovat ji. Proto tuto část rozdělím na dvě části, první bude zaměřena na samotné vykreslování a druhá na práci s mapou (posouvání a zvětšování).

### 4.2.1 Vykreslování

Jelikož se všechny zobrazené prvky na mapě v průběhu celé hry mění, musel jsem i já vykreslovat tyto prvky dynamicky. K tomuto účelu jsem použil třídu *hud*, jež patří do základní kolekce tříd Shiva engine. Tato třída obsahuje veškeré funkce, které nabízí HUD editor. Třída obsahuje funkce pro vykreslení různých objektů, jako jsou obrázky, tlačítka posuvníky, atd., nastavování jim různých parametrů od umístění na obrazovce až po průhlednost. Kromě mnoha parametrů se jim dají nastavit i akce. Mimo tyto funkce obsahuje také funkce pro přehrávání zvuků a videa.

Jak už bylo řečeno, každá úroveň obsahuje strukturu, jež byla popsána výše. Tato struktura se musí projít a každý objekt zvlášť vykreslit. U listů, květů, plodů, dále jen subelementy a u skal, zdrojů vody a půdy dále jen mapelementy, stačí projít jednotlivé části XML postupně a u každého tohoto objektu zavolat metodu na vykreslení. U větví a kořenů, dále jen elementů, je potřeba zajistit, aby se rekurzivně volala metoda, která vykreslí jeden element a zároveň vyvolá vykreslení potomků tohoto elementu.

Každý objekt na mapě je vykreslen zvlášť jako samotný obrázek a je u něj nastavena pozice, šířka, výška, úhel natočení a samozřejmě název obrázku, který se má vykreslit. Nastavení těchto parametrů stačí k tomu, aby se daný objekt na mapě vykreslil správně. Abych mohl tyto parametry nastavit správně a tím vznikla mapa celého světa, potřebuji pár pomocných parametrů.

U map elementu si vystačím s typem map elementu, uloženým v atributu *Category*, pomocí něhož nastavím název obrázku, který se má použít a šířku i výšku daného obrázku. Pro zjištění pozice slouží atributy *StartX*, *StartY*.

K vykreslování stromu, aby jednotlivé elementy na sebe navazovaly a aby subelementy doléhali k větvím a tím vzniknul celistvý strom, už potřebuji parametrů trochu více. Těmi jsou *IDTree*, ke zjištění zdali se jedná o hráčův nebo nepřítelův strom, *TypeElement*, zdali je to element nebo subelement. U elementů potřebuji znát *MasterType*, ze kterého určím, jestli se jedná o kořen nebo větev. A u subelementů *Position*, jež říká na jaké straně větve se daný subelement nachází. Také potřebuji znát úhel natočení, ten získam z atributu *Angle*. Všechny tyto parametry jsou uloženy v XML, a proto je stačí načíst a předat metodě na vykreslování, ve které už z těchto parametrů nastavím hlavní parametry obrázku.

Zde jde vidět jak se v Shiva 3D vykreslí obrázek se základními parametry.

```
hud.newComponent ( this.getUser ( ), hud.kComponentTypePicture, HUDname )

local hPicture = hud.getComponent (  this.getUser ( ), HUDname )

hud.setComponentPosition (hPicture, StartX, StartY )
hud.setComponentBorderColor (hPicture,0,0,0,0)
hud.setComponentSize ( hBranch, this.nMapElementWidth ( ), this.nMapElementHeight ( ))
hud.setComponentRotation (hPicture, Angle )
hud.setComponentBackgroundImage (hPicture, imageName )
```

```
hud.setComponentOrigin (hPicture, hud.kOriginCenter)
```

Tímto kódem se vykreslí obrázek tak, že střed obrázku bude na souřadnicích *StartX*, *StartY*. Tento systém je použit u elementů a mapelementů. U subelementů jsem narazil na problém, kdy se subelementy vykreslili na správné straně i pod správným úhlem, jež jsem nastavil, ale byly o nepatrný kousíček posunuty. Jejich hrany, které se měly dotýkat a díky čemuž potom subelementy vypadají jako zrcadlově otočeny, se spolu vůbec nedotýkaly. Tohle jsem ale vyřešil díky správnému nastavení *hud.setComponentOrigin,* jež slouží k nastavení středové pozice obrázku. V následující ukázce kódu se podle pozice subelementu a jeho náklonu nastavují správné středové pozice.

```
if Position == 1  -- left
then
        hud.setComponentRotation (hPicture, Angle - 90   )
        if Angle == -180
        then
                hud.setComponentRotation (hPicture, Angle +90  )
        end
        if Angle == 0 or Angle == 360 or Angle == -360 or Angle == -180
        then
                hud.setComponentOrigin (hPicture, hud.kOriginRight)
        elseif Angle == 90 or Angle == 450 or Angle == -270
        then
                hud.setComponentOrigin (hPicture, hud.kOriginBottom)
...
```

HUD editor rozděluje obrazovku na rozměry 100x100 a nezáleží jaké rozlišení má displej. Vždy vezme rozlišení zařízení a pro něj spočítá, na kolik pixelů se bude zobrazovat bod o velikosti 1x1 z HUD editoru. Pro iPhone, který má poměr stran 3/2, tj. rozlišení 480x320, bude bod o velikosti 1x1 z HUD editoru, zabírat plochu 4,8x3,2 pixelů na iPhone. Pokud se, ale změní poměr stran, to může nastat, pokud se změní zařízení např. na iPad, jež má poměr stran 4/3, tj. rozlišení 1024x768, bude i tento jeden bod z HUD editoru zachovávat tento poměr stran. A pokud jsme vykreslovali nějaký obrázek a on vypadal dobře na iPhone, na iPadu už bude deformovaný. K tomu aby se tomuto nešvaru předešlo, existuje nastavení, po němž si obrázek bude udržovat stejný počet stran nehledě na rozlišení displeje. Onou funkcí je *hud.setComponentAspectInvariant (hComponent, bInvariant )*.

### 4.2.2 Posouvání a zvětšování

Jelikož naše mapa zasahuje i mimo obrazovku, je jasné, že musely být implementovány funkce pro práci s ní. Těmito funkcemi jsou posouvání a zvětšování. K tomu aby se mapa posunula nebo změnila velikost, bych musel každý objekt na mapě zvlášť posouvat nebo mu měnit velikost. To je ale dost neefektivní, naštěstí třída *hud* obsahuje funkci *hud.setComponentContainer ( hComponent,*

*hContainer )*, která nastaví objektu tzv. kontejner. Když se poté manipuluje s tímto kontejnerem, tak se operace aplikující na něj aplikují i na objekty, které tento kontejner obsahuje. Proto jsem všechny objekty, jež jsem vykreslil na mapě, přidal do jednoho kontejneru, díky čemuž už s mapou mohu pracovat jako s jedním objektem.

Pro zjišťování dotyku na obrazovce existují v Shiva 3D tři handlery. Prvním z nich je *onTouchSequenceBegin*, ten se spustí, pokud se uživatel dotkne displeje. Zde kontroluji, zda je zobrazena mapa a poté povolím nebo nepovolím operace s její manipulací.

V druhém handleru se už provádí samotné operace s mapou. Handler se jmenuje *onTouchSequenceChange* a spouští se, pokud se prst(y) na displeji pohybují. Jako parametry předává počet prstů, jež se displeje dotýkají a pro každý prst jeho aktuální pozici na displeji. Nejdříve si načtu do lokálních proměnných souřadnice, kde se nacházely prsty naposledy. Poté uložím do globálních proměnných souřadnice, kde se právě prsty nachází, které využiji při dalším posunu jako minulé souřadnice dotyku. Když již znám aktuální pozice prstů a minulé pozice prstů, nic mi nebrání, abych mohl mapu posouvat a zvětšovat.

Jestliže se uživatel dotýká jedním prstem na mapě, mohu tuto mapu posouvat. Pokud od aktuální pozice odečtu pozici minulou, získám vzdálenost, o kterou mám mapu posunout. Teď již stačí zkontrolovat, jestli se tímto posunem nedostanu přes hranici mapy. Pokud je tato podmínka splněna, stačí mapě nastavit nové souřadnice, které se získají přičtením vzdálenosti pro posun k minulým souřadnicím mapy.

Zde je vidět implementace, kde se počítá posun mapy, kontrolují hranice mapy a následně se mapa posouvá.

```
local nScroolX = ( nX0 - nLastX0 ) * 50
local nScroolY = ( nY0 - nLastY0 ) * 50
if this.nMapWidth ( ) > 100 and this.nMapHeight ( ) > 100
then
        if ( this.nMapCenterX ( ) - nScroolX) < (this.nMapWidth ( ) - 50) and ( this.nMapCenterX
( ) - nScroolX) > 50 and ( this.nMapCenterY ( ) - nScroolY) < (this.nMapHeight ( ) - 50) and (
this.nMapCenterY ( ) - nScroolY) > 50
        then
                this.nMapCenterX ( this.nMapCenterX ( ) - nScroolX)
                this.nMapCenterY ( this.nMapCenterY ( ) - nScroolY)
                local hComponent = hud.getComponent ( this.getUser ( ), "MapContainer" )
                local nPositionX, nPositionY = hud.getComponentPosition ( hComponent )
                nPositionX = nPositionX + nScroolX
                nPositionY = nPositionY + nScroolY

                hud.setComponentPosition( hComponent, nPositionX, nPositionY)
        end
end
```

Pokud se už uživatel dotýká dvěma prsty, mohu mapu zvětšovat. K tomu potřebuji znát vzdálenost prstů od sebe a to jak minulou vzdálenost, tak i aktuální. Jejich rozdílem potom získám vzdálenost, o jakou mám mapu zvětšit.

Každý prst si mohu představit jako jeden bod na rovině. K určení jejich vzdálenosti stačí spočítat velikost vektoru, jež je dán dvěma body, respektive prsty. Abych mohl spočítat velikost vektoru, potřebuji nejdříve souřadnice vektoru [2].

Výpočet souřadnic vektoru minulého postavení prstů *Ulast* získáme ze vztahu

$$Ulast[x] = Last2[x] - Last1[x] \tag{2}$$

$$Ulast[y] = Last2[y] - Last1[y] \tag{3}$$

kde *Last1* jsou minulé souřadnice prvního prstu a *Last2* minulé souřadnice druhého prstu.

Výpočet souřadnic vektoru aktuálního postavení prstů *Unew* spočítáme následovně

$$Unew[x] = New2[x] - New1[x] \tag{4}$$

$$Unew[y] = New2[y] - New1[y] \tag{5}$$

kde *New1* jsou aktuální souřadnice prvního prstu a *New2* aktuální souřadnice druhého prstu.

Teď když znám souřadnice obou vektorů, mohu spočítat jejich velikost a tím dostat vzdálenosti prstů od sebe. Velikost vektoru *Ulast* a *Unew* získáme následovně[2]:

$$|Ulast| = \sqrt{Ulast[x]^2 + Ulast[y]^2} \tag{6}$$

$$|Unew| = \sqrt{Unew[x]^2 + Unew[y]^2} \tag{7}$$

Dalším krokem je zjištění rozdílů jejich délek, ke zjištění velikosti o jakou mám mapu zvětšit. K tomu potřebuji velikost vektoru, jehož souřadnice jsou šířka a výška mapy a tuto velikost rozšířím o rozdíl délek vektorů danými prsty. A spočítám nové velikosti šířky a výšky mapy pomocí sinovy věty [12].

Velikost vektoru minulé velikosti mapy *UmapOld* získáme ze vztahu

$$|UmapOld| = \sqrt{WidthOld^2 + HeightOld^2} \tag{8}$$

kde *WidthOld* je minulá šířka mapy (x-ová souřadnice vektoru mapy) a *HeightOld* je minulá výška mapy (y-ová souřadnice mapy).

Pokud velikost zvětšení je dána vzorcem

$$Distance = Ulast - Unew \tag{9}$$

získáme velikost vektoru aktuální velikosti mapy *UmapNew* následovně

$$|UmapNew| = |UmapOld| - Distance \tag{10}$$

Jestliže velikosti úhlů se získají ze vztahu

$$\sin\alpha = HeightOld/|UmapOld| \tag{11}$$

$$sin\,\beta = WidthOld/|UmapOld| \tag{12}$$

spočítání nových délek šířky *WidthNew* a výšky *HeightNew* mapy pomocí sinovy věty se získá následovně

$$WidthNew = (|UmapNew| * sin\,\alpha)/sin\,90 \tag{13}$$

$$HeightNew = (|UmapNew| * sin\,\beta)/sin\,90 \tag{14}$$

Teď když už když znám nové velikosti mapy, stačí zkontrolovat, zdali nejsou menší než velikost displeje a zdali nejsou větší než maximální přiblížení. Pokud splňují tyto podmínky, nic mi nebrání nastavení těchto nových rozměrů mapě. Naštěstí Shiva engine obsahuje matematické funkce na spočítání vektoru atd., takže v samotné implementaci to už nevypadá tak hrozivě. Celý výpočet se provede pomocí pár řádků kódu.

```
local nPreviousAxisBetweenFingersX, nPreviousAxisBetweenFingersY = math.vectorSubtract ( nLastX1, nLastY1, 0, nLastX0, nLastY0, 0 )
local nNewAxisBetweenFingersX, nNewAxisBetweenFingersY = math.vectorSubtract ( nX1, nY1, 0, nX0, nY0, 0 )

local nDeltaDist = math.vectorLength ( nPreviousAxisBetweenFingersX, nPreviousAxisBetweenFingersY, 0 ) - math.vectorLength ( nNewAxisBetweenFingersX, nNewAxisBetweenFingersY, 0 )

local hComponent = hud.getComponent ( this.getUser ( ),  "MapContainer")
local nSizeX, nSizeY = hud.getComponentSize ( hComponent )
```

```
local nSizeXNew, nSizeYNew = math.vectorSetLength ( nSizeX, nSizeY, 0, math.vectorLength ( nSizeX, nSizeY, 0 ) - nDeltaDist * 50 )

if (width > 100 and height > 100) and (width - nSizeXNew < 199 and height - nSizeYNew < 199) then
	hud.setComponentSize ( hComponent, nSizeXNew, nSizeYNew )
end
```

V této ukázce se spočítají všechny velikosti vektorů i velikost, o kterou se má mapa zvětšit. Také se zkontroluje, jestli nová velikost mapy není menší než minimální velikost mapy a větší než maximální velikost. Následně se mapa zvětší.

## 4.3 Stavění

Po stisknutí tlačítka pro stavění se otevře stavěcí menu, které nabízí postavení třech druhů větví/kořenů, listu a květu. Pod každým (sub)elementem se zobrazuje hodnota, která udává, kolik kusů (sub)elementu mohu postavit s aktuální energií. Zjišťuje se zde, které (sub)elementy lze postavit a podle toho se nastaví textura tlačítek v menu. Pokud je energie menší než cena (sub)elementu, nastaví se obrázek s bílou texturou, přes který je znak dollaru a další kontroly neprobíhají. Na obrázku Obr. 11 je zobrazeno stavěcí menu, se všemi druhy stavěcích textur.


*Obr. 11 - Stavěcí menu*

Pokud je ale energie dostatek začínají se provádět kontroly s volnými pozicemi a vypočítávají se kolize. Před nimi se v XML najde element, ve kterém je vážka podle ID elementu, jež se nastavuje při průletu do dalšího elementu.

Kontrola volných pozic se prování u subelementů. Když známe element, ve kterém se vážka nachází a na kterém chceme stavět, zjistíme si z atributu *MasterType*, zdali jsme v kořenu nebo větvi. Pokud jsme v kořenu, což odpovídá hodnotě 2 u *MasterType*, můžeme nastavit textury s červenými obrázky subelementů. Když jsme ale ve větvi, což odpovídá hodnotě 1 u *MasterType*, přejdeme na hledání volných pozic.

Jelikož známe element s vážkou, zjistíme si hodnoty startovních bodů (*StartX*, *StartY*) a koncových bodů (*EndX(1-3)*, *EndY(1-3)*). Dále začneme v XML procházet všechny subelementy hráčova stromu. U každého subelementu kontrolujeme, zdali se jeho startovní pozice nerovnají startovní

nebo koncové pozici elementu, ve kterém se nacházíme. Pokud se rovnají, ještě to neznamená, že tato pozice je obsazena. Na jedněch souřadnicích mohou být dva subelementy, jeden na pravé straně větve a druhý na levé straně větve. Na které straně se subelement nachází je uloženo v atributu *Position*, ten nabývá hodnot 1 pro levou stranu a 2 pro pravou stranu. O tuto hodnotu zvýšíme jednu ze čtyř proměnných, podle toho, na které pozici se souřadnice rovnaly a pokračujeme kontrolou dalšího subelementu. Samozřejmě si na začátku procházení všech subelementů nastavím tyto hodnoty na nulu.

Tímto způsobem si naplníme čtyři proměnné, podle kterých zjistíme, zdali existují volné pozice a kde se nachází. Zjistíme to jednoduše tak, že pokud je v proměnné uložena hodnota 3, tak na této pozici jsou postaveny dva subelementy a nový tam postavit nelze. Pokud je tam hodnota menší než 3, znamená to, že jedna nebo obě pozice jsou volné. Když takhle zkontrolujeme všechny čtyři proměnné, zjistíme, zda je nějaká pozice na našem elementu volná. Jestliže se nachází alespoň jedna volná pozice, nastavíme listu i květu textury pro stavění, pokud jsou, ale všechny obsazeny nastavíme červené textury.

Kolize se počítají u elementů. Nejdříve se projdou všichni potomci elementu, ve kterém se nacházíme. Pokud je to element typu 0, tj. jedná se o zátku, můžeme na něm postupně zkusit postavit všechny tří typy elementů a u nich zkusíme, jestli s něčím nekolidují. To se provede tak, že si u naší zátky zjistíme startovní pozice a úhel naklonění. Z těchto hodnot si postupně spočítáme koncové souřadnice všech elementů, které se zkouší postavit. Jelikož délky jednotlivých částí větví, dále jen stroků, i úhly mezi nimi jsou pevně dány je toto spočítání triviální přes sinovu větu, podobně jak tomu bylo u posouvání mapy, akorát teď známe délky i úhly a jednu souřadnici a snažíme se spočítat druhou souřadnici.

Když máme spočítané souřadnice konců jednotlivých elementů, zkusíme nejdříve, jestli nejsou větší než hranice mapy a u kořenů jestli nevylézají z půdy a jestli větve nerostou do půdy. Poté vyzkoušíme, jestli nekolidují s ostatními větvemi nebo skálami.

Aby se nemusely procházet všechny mapelementy a stroky, tak se pří startu nové úrovně vytvoří mřížka mapy. Cela herní plocha se rozdělí na 250x250 = 62 500 poliček šachovnice. Jedno šachové políčko má tedy délku 1. Každé políčko je jedna hodnota v tabulce. Políčko šachovnice nabývá tří typu hodnot:

- *(0) volno*
- *(1) obsazeno elementem nebo skálou*
- *(ID zdroje) obsazeno zdrojem*

Po té když se kontrolují kolize, stačí zkontrolovat, zda jsou některé pozice na mřížce, přes které chceme postavit novy element obsazené. Políčko šachovnice se povazuje za překryté pokud se na něm samém nebo na jeho sousedovi nachází stroke nebo skála. Element typu 1, který by byl horizontálně, by měl tedy výšku 8 poliček a sirku 3.

Takhle vyzkoušíme na každé zátce postavit všechny tří elementy, a pokud je alespoň jedna zátka u jednoho elementu, na kterou ho lze postavit, nastavíme mu texturu pro stavění, pokud ne, nastavíme červenou texturu.

Pokud si hráč vybere (sub)element, který chce postavit, stavěcí menu se skryje. Na všech volných a nekolidujících pozicích elementu, ve kterém se nachází, se vykreslí vybraný (sub)element s bílou texturou dále jen bílý element. Ty indikují možná místa, kde lze (sub)element umístit a také kam až bude zasahovat. To jak se zjistí volné pozice a nekolidující element jsme si ukázali o pár řádků výše. A to jak se vykreslí daný (sub)element na určitou pozici jsme si ukázali v kapitole 4.2.1 Vykreslování, jediný rozdíl je ten, že dané (sub)elementy jsou pouze dočasné a po ukončení stavění se smažou. Nad hráčovým prstem se ještě vykreslí vybraný (sub)element s modrou texturou. Ten zůstává nad prstem po celou bodu dotyku displeje.

Při pohybu (sub)elementu (prstu) se vypočítává jeho vzdálenost od všech možných míst umístění. Tato vzdálenost se počítá přes velikost vektoru, jenž je daný dvěma body. Jedním je pozice prstu a druhým je pozice bílého elementu. Jak se taková vzdálenost spočítá, jsme si ukázali v kapitole 4.2.2. Posouvání a zvětšování.

Pokud je tato vzdálenost od všech bílých elementů větší než minimální vzdálenost pro stavění, zůstávají bílé elementy nezměněny. Jakmile je menší, zjistí se, u kterého je nejmenší a tomu se změní textura z bílé na červenou, aby byli dobře odlišitelné. Po ukončení dotyku se smažou bílé elementy a na místo, kde byl element s červenou texturou, se postaví patřičný (sub)element. Když se takhle postaví (sub)element, přidá se nejen do mapy, ale také do XML struktury.

Přidání subelementu do naší XML struktury je vcelku jednoduché. Jelikož známe všechny jeho atributy (jsou totožné s červeným subelementem), stačí v XML struktuře přidat do elementu SubElements u stromu, na kterém se staví, nový element s těmito atributy.

Zde je ukázka kódu jak se přidá nový element s danými atributy do nějakého jiného elementu.

```
local hXMLSubElement=xml.appendElementChild ( subElements, "SubElement", "" )
hXMLAtribute = xml.appendElementAttribute ( hXMLSubElement, "StartX", string.format (
"%5.5f", StartX ) )
hXMLAtribute = xml.appendElementAttribute ( hXMLSubElement, "StartY", string.format (
"%5.5f", StartY ) )
hXMLAtribute = xml.appendElementAttribute ( hXMLSubElement, "Angle", string.format (
"%5.5f", Angle ) )
hXMLAtribute = xml.appendElementAttribute ( hXMLSubElement, "Type", string.format (
"%0.5i", Type ) )
hXMLAtribute = xml.appendElementAttribute ( hXMLSubElement, "Position", string.format (
"%0.1i", Position) )
```

Přidání elementu je o něco složitější. Každý nový element se totiž přidává na místo zátkového elementu a také pro všechny jeho koncové body musí vytvořit nové zátkové elementy, jež budou jeho dětmi. Známe element, na kterém stavíme, známe všechny atributy nového elementu, jedině neznáme zátkový element, který nahrazujeme. To, ale není nic strašného, jednoduše projdeme všechny zátkové elementy a porovnáme jejich startovní pozice se startovními pozicemi nově budovaného elementu. Najdeme-li shodu, našli jsme náš zátkový element. Nyní mu stačí upravit atributy koncových bodů a atribut určující typ elementu.

Jelikož už se nejedná o zátkový element, ale element, který se skládá z několika stroků, musíme tyto nové stroky také přidat do XML. Je to jako u subelementů. Do elementu *Strokes* u stromu, na kterém se staví, přidáme nové elementy *Stroke* s atributy podle nově postaveného elementu.

Teď už máme nahrazený zátkový element, i zapsané jednotlivé stroky, zbývá poslední krok a tím je vytvoření nových zátkových elementů na nově postaveném elementu. Provede se to tak, že nově postavenému elementu přidáme další elementy, jež nastavíme na zátkové elementy, tj. atribut *TypeElement* na hodnotu 0 a jejich startovní pozice se budou rovnat koncovým pozicím nově postaveného elementu.

## 4.4 Obtížnost úrovní

Obtížnost jednotlivých úrovní je specifikována několika parametry. Startovní rozestavění a velikost jednotlivých stromů a mapelementů také určuje obtížnost úrovně. Tomu se ale věnuje kapitola 4.6 Editor úrovní.

Prvními parametry jsou tzv. časová nastavení. Prvním z nich je nastavení přeměny květů na plody. Tj. času, za který se květy přemění na plody. Dalším takovým je růst konkurenčních stromů. Ten určuje časový úsek, za který konkurenční stromy postaví nějaký nový (sub)element. Mimo ně existuje i takový, jenž určuje, v jakých časových intervalech bude strom čerpat zdroje. A poslední je pro určení údržby stromu. Ten říká, v jakých intervalech si bude strom odebírat energii nutnou pro svoji údržbu.

K tomu abychom ve hře mohli pracovat s časem, existují dvě metody. První je vytvoření a nastavení časovače (timer) a následná práce s ním. Tato metoda je trochu komplikovanější a náročnější na práci než je tomu u jiných programovacích jazyků. Proto jsem zvolil metodu číslo dvě. Shiva engine obsahuje funkci, která dovoluje vykonávat události za určitý čas. Onou funkcí je *this.postEvent ()* s parametry čas, název události a dalšími parametry, jež jsou předány dané události. V našem případě si stačí vytvořit handler ve kterém na začátku specifikujeme ať se za určitý čas zase tento handler vykoná. Takhle máme zajištěno, že se tento handler bude vykonávat v pravidelných intervalech. Díky tomu můžeme jednoduše zajistit, ať se za námi specifikovaný čas např. změní květy na plody.

V našem případě jsem si vytvořil jeden handler, který se spouští v krátkých časových intervalech. Při každém tomto intervalu se o tento interval zvýší hodnota atributů, které poté porovnávám

s námi definovanými časovými nastaveními. Pokud je hodnota atributu větší nebo rovna našemu časovému nastavení, vynuluji tento atribut a provedu kód, jenž vykoná např. přeměnu květů na plody nebo údržbu stromu podle toho jaké atributy jsme porovnávali.

Zde je ukázka našeho handleru, který volá sám sebe každou 0.1s, pokud není hra pozastavena a pokud nastane čas, kdy mají růst konkurenční stromy, zavolá se funkce, která zajišťuje tento růst.

```
local increment = 0.1
if (this.bShowMenu ( ) ~= true) and (this.bShowMessage ( ) ~= true)
then
        this.postEvent ( increment, "onTimer" )

         this.nTimerCompetitors ( this.nTimerCompetitors ( ) + increment)
         this.nTimerSuck ( this.nTimerSuck ( ) + increment)
         this.nTimerCondition ( this.nTimerCondition ( ) + increment)
         this.nTime ( this.nTime ( ) + increment)

        if (this.nTimerCompetitors ( ) >= this.nTimeGrowCompetitors ( ))
        then
                this.nTimerCompetitors ( 0 )
                this.GrowCompetitors ( )
        end
        …
end
```

Mezi další nastavení patří nastavení počtu plodů pro výhru. Ve hře se poté vždy po postavení plodu (přeměně květu na plod) kontroluje, zdali není počet postavených plodů větší nebo roven této hodnotě. Pokud je, tak se úroveň ukončí a hráč je informován o výhře úrovně. Pokud máme nastavení pro výhru úrovně, musíme mít také nastavení pro prohru. Jelikož je úroveň prohraná pokud se s energií stromu dostaneme pod nějakou hranici, existuje nastavení, jež určuje tuto hranici. Ve hře se při každé aktualizaci energie kontroluje, jestli energie není pod touto hranicí a pokud tomu tak je, hráč úroveň prohrává a je o tom informován.

Jak už bylo řečeno výše, nastavovat se dají i jednotlivé váhy zdrojů, díky kterým poté každý zdroj přidává méně nebo více energie. U zdrojů kromě vah existuje nastavení, jež specifikuje, kolik zdroje se za jeden časový úsek čerpání zdrojů vyčerpá. O tom více v kapitole 4.5 Práce se zdroji.

Také lze nastavovat ceny jednotlivých částí stromu a kolik potřebují energie pro údržbu. Je přítomno i nastavení pro počáteční množství energie každého stromu.

Mezi poslední nastavení patří umělá inteligence konkurenčních stromů. To se skládá ze dvou nastavení. První je čerpací koeficient, jenž určuje, kolik procent zdrojů při jejich čerpání bude mít strom k dispozici. Toto nastavení je taková náhrada za vážku, jelikož v konkurenčních stromech žádná nejezdí. A díky tomu nemůže sbírat natěžené zdroje a předávat je k dispozici svému stromu v podobě energie. Druhým a zároveň i posledním nastavením je agrese konkurenčních stromů.

Toto nastavení určuje, jak moc budou konkurenční stromy obléhat hráčův strom. Tzn., jak moc budou hráčův strom obrůstat, aby neměl možnost postavit další elementy.

Při růstu konkurenčních stromů se nejprve vygeneruje náhodné číslo, podle kterého se poté určí, zda se bude stavět normální element nebo subelement. Pokud se má postavit element, rozhodne mezi třemi akcemi, jež specifikují postavení onoho elementu.

- *Útok na hráčův strom, tj. jeho obrůstání.*
- *Honba za zdroji v půdě.*
- *Honba za světlem.*

Pro každou tuto akci se určují souřadnice, které ji specifikují a ke kterým se bude strom snažit dostat při postavení tohoto elementu. Těmto souřadnicím budeme říkat target. Pro určení targetu se vygeneruje další náhodné číslo v rozmezí 0 až 100 a porovná se s nastavením agrese, jež může nabývat také hodnot 0 až 100. Pokud je agrese větší znamená to, že se při tomto postavení elementu bude konkurenční strom snažit obrůst hráčův strom. A jako target si vezme startovní souřadnice hráčova stromu. Když je agrese menší než náhodně vygenerované číslo, rozhoduje se, mezi honbou za světlem a honbou za zdroji v půdě. Při honbě za světlem se jako target nastaví nějaká náhodná souřadnice ve vzduchu pro rozšíření koruny stromu.

Při honbě za zdroji v půdě se jako target určí nějaká startovní souřadnice vody nebo půdy. Ty se určí tak, že se v XML náhodně vybere nějaký mapelement. Z něj si zjistíme, zdali se jedná o skálu nebo zdroj. Pokud je to zdroj (voda nebo půda), zjistíme si startovní pozice a ty nastavíme jako target. Když se jedná o skálu, vybereme si náhodně další mapelement a postup opakujeme. Aby toto hledání vody nebo půdy nezabíralo mnoho času, vybíráme maximálně deset mapelementů a pokud i desátý je skálou, přejdeme na hon za světlem. Při honbě za světlem víme, že budeme stavět větev, při honbě za zdroji z půdy víme, že budeme stavět kořen. Ale při útoku na hráčův strom můžeme postavit jak kořen, tak i větev. K tomuto rozhodnutí opět využijeme náhodný generátor, jenž určí, zdali budeme stavět větev nebo kořen dále jen mastertype. Určení targetu se provádí pomocí tohoto kódu:

```
if (math.random ( 0, 100 )<this.nOpponentAgresion ( ))
then  --útok na hráčův strom
        return math.roundToNearestInteger ( math.random ( 0.6,3.4 ) ), this.PlayersTreeX(),
this.PlayersTreeY(), math.roundToNearestInteger ( math.random ( 1.3, 2 ) ), false

elseif (math.random ( 0, 100 )<50)
then  -- honba za zdroji v půdě
        index = math.roundToNearestInteger ( math.random ( 1, xml.getElementChildCount (
MapElements ) )-1 )
         MapElement = xml.getElementChildAt ( MapElements, index )

        while ((MapElement and Category>2) and (j<10) )
         do -- vyhledaní vody nebo půdy
                j=j+1
```

```
                    hXMLAtribute = xml.getElementAttributeWithName ( MapElement, "Category" )
                    Category = string.toNumber ( xml.getAttributeValue ( hXMLAtribute ) )
                    index = math.roundToNearestInteger ( math.random ( 1,
xml.getElementChildCount ( MapElements ) )-1 )
                    MapElement = xml.getElementChildAt ( MapElements, index )
            end

           if MapElement and (Category<=2)
           then
                    return math.roundToNearestInteger ( math.random ( 1.3,3.4 ) ), StartX, StartY, 2,
false
            else -- honba za světlem, 10krat nalezená skála
                    return math.roundToNearestInteger ( math.random ( 0.6,3.4 ) ), math.random (
0,200 ), math.random ( 0,200 ), 1, false
            end

 else -- honba za světlem
           return math.roundToNearestInteger ( math.random ( 0.6,3.4 ) ), math.random ( 0,200 ),
math.random ( 0,200 ), 1, false
end
```

Když máme určený target i mastertype, vybereme náhodně jeden ze tří typů elementů a zkoušíme nalézt zátku, na kterou tento element postavíme. Tato zátka musí být nejblíže našemu targetu a musí jít na ni tento element postavit. Začínáme u prvního elementu konkurenčního stromu. Pro všechny jeho potomky spočítáme jejich vzdálenost od našeho targetu. Tu spočítáme přes velikost vektoru, jež je určen targetem a startovními souřadnicemi elementu. Jak se tato velikost spočítá je popsáno v kapitole 4.2.2. Posouvání a zvětšování.

Pro element jehož mastertype je totožný s tím, který chceme postavit a jenž má nejmenší vzdálenost od targetu, zjistíme, jestli se jedná o zátku. Pokud se o zátku nejedná, procházíme všechny jeho potomky a aplikujeme stejný postup, dokud na zátku nenarazíme. Na takhle nalezeném zátkovém elementu zjistíme, jestli na něm můžeme náš element postavit. Pokud postavit lze, jednoduše ho postavíme. Zjišťování zdali se daný element dá postavit a jak se postaví je popsáno v kapitole 4.3 Stavění.

Když ale postavit nelze, přejdeme ze stavění elementu na stavění subelementu. Předtím, ale přidáme tomuto elementu nový speciální atribut, který říká, že na něm nelze stavět. Samozřejmě se při procházení potomků jednotlivých elementů kontroluje, zdali neobsahuje tento speciální atribut. Pokud, ale tento speciální atribut obsahují všichni potomci elementu, přiřadíme ho i tomuto elementu. Tím se zabrání, aby se při dalším stavění nepočítala vzdálenost elementu, na který nelze stavět, a který by měl tuto vzdálenost nejmenší, pokud by se jednalo o target, jenž už se zkoušel.

Vyhledání nejlepší zátky pro postavení se provádí tímto kódem:

```
if (TypeCurrentElement==0)
```

```
then
        if (this.CheckGrowthPossibility ( hXMLElement, TypeElement )==true)
        then
                hXMLBestElement=hXMLElement
                Possible=true
        else
                Possible=false
                xml.appendElementAttribute ( hXMLElement, "OpenPathForAI", "0")
        end
else
        hChild = xml.getElementFirstChild ( hXMLElement )
        while (hChild) do
                hXMLAtribute=xml.getElementAttributeWithName ( hChild, "OpenPathForAI" )
                if not (hXMLAtribute) then
                        --vypočtení vzálenosti a urcení elementu s nejmensí vzdáleností
                        ...
                end
        hChild=xml.getElementNextSibling ( hChild )
        end
        if (hXMLTempElement)
        then --element s nejmenší vzdáleností
                hXMLBestElement, Possible, TypeElement  = this.GetBestBuildElement (
hXMLTempElement, TypeElement, TargetX, TargetY, TargetMasterType, hXMLLevel )
         else
                xml.appendElementAttribute ( hXMLElement, "OpenPathForAI", "0")
                Possible=false
        end
end
return hXMLBestElement, Possible, TypeElement
```

Takto se u konkurenčního stromu postaví nový element. Pro stavění subelementů se vyhledá náhodně nějaká větev nebo zátka u větve. Na jejích startovních souřadnicích se vyzkouší, zdali se dá postavit nějaký subelement. Pokud ano, postaví se na jedné z volných pozic list nebo květ, jež je náhodně vybrán. Když jsou obě pozice zabrány, nepostaví se vůbec žádný subelement.

Situace kdy se nepostaví žádný (sub)element je zde proto, aby se zabránilo zbytečně dlouhému hledání volné pozice, které by hru zpomalovalo. Šance, že se tohle opravdu přihodí, je ale tak malá, že růst konkurenčních stromů nijak neovlivní a lze ji lze zanedbat.

## 4.5 Práce se zdroji

Pod tímto názvem se ukrývá samotné těžení zdrojů. Z předchozích kapitol víme, že existují čtyři druhy zdrojů. Také víme, že tam kde je list, se těží světlo, kde se kořen prolíná s vodou nebo půdou se těží voda nebo půda. A že se vzduch těží ve všech větvích.

K tomu aby se natěžené zdroje zobrazily uvnitř stromu, se používají speciální tabulky tzv. *hashtable*. Pro každý zdroj existuje jedna hashovací tabulka. Jako klíč je dáno *IDStroke*, pod kterým je uložena hodnota natěženého zdroje v tomto stroku. Kromě těchto zdrojových hashovacích tabulek, dále jen zdrojové tabulky, existuje ještě několik dalších hashovacích tabulek, tzv. pomocné tabulky. Díky nim se pří každém čerpacím cyklu správně naplní zdrojové tabulky, pomocí kterých se zobrazují natěžené zdroje uvnitř stromu. Zdrojové tabulky existují jenom pro hráčův strom, ale pomocné tabulky existují i pro konkurenční stromy.

Při postavení nového (sub)elementu se vždy vyzkouší, jestli se v nějakém novém nebo stávajícím stroku netěží nějaký zdroj a naplní se pomocné tabulky. Při postavení nového kořenu se pro každý jeho stroke vyzkouší, zdali se neprotíná s vodou nebo půdou. Kontrola probíhá pomocí mřížky na mapě, jak bylo popsáno v kapitole 4.3 Stavení. Pokud se některé stroky dotýkají vody nebo půdy, přidají se jejich ID jako klíče do pomocné tabulky *hashTree(1-3)StrokesAreas*, podle toho k jakému stromu kořen patří. A do jejich hodnot se uloží ID oblasti (*IDArea*), ze které čerpají. Jeden stroke může čerpat maximálně ze tří oblastí.

Hashovací tabulky v Shiva 3D nefungují jako hashovaní tabulky v jiných programovacích jazycích, kdy je možno pod jeden klíč uložit více hodnot, ale spíše jako slovníky, kdy je pod jeden klíč možno uložit jenom jednu hodnotu. K tomu aby bylo možno uložit více oblastí pod jeden stroke musí být hodnota pod tímto klíčem zapsána ve speciálním tvaru, kde každé ID oblasti je odděleno středníkem. Následný zápis do této pomocné tabulky pro hráčův strom vypadá v kódu takto:

```
local firstArea, secondArea, thirdArea = this.GetSubString ( sIDStroke, 1, "area" )
if (firstArea == nil)
then
        hashtable.add ( this.hashTree1StrokesAreas ( ), sIDStroke , string.format ( "%0.2i",
sIDArea )..";;;" )
elseif secondArea == nil
then
        hashtable.set ( this.hashTree1StrokesAreas ( ), sIDStroke ,  string.format ( "%0.2i",
firstArea ) ..";"..string.format ( "%0.2i", sIDArea ) ..";;" )
elseif thirdArea == nil
then
        hashtable.set ( this.hashTree1StrokesAreas ( ), sIDStroke ,  string.format ( "%0.2i",
firstArea ) ..";".. string.format ( "%0.2i", secondArea )..";"..string.format ( "%0.2i", sIDArea ) ..";" )
end
```

Pro čerpání zdrojů z půdy existují kromě těchto pomocných tabulek i další pomocné tabulky. Tyto pomocné tabulky se naplňují vždy na začátku úrovně a specifikují jednotlivé oblasti. Jako klíče jsou uloženy všechny ID oblastí a v hodnotách pro tyto klíče jsou specifické vlastnosti této oblasti pro danou pomocnou tabulku. Např. pro pomocnou tabulku *hashMapAreaCategory* jsou v hodnotách klíčů uloženy hodnoty kategorií, do kterých dané oblasti patří.

Jelikož na jednom stroku mohou být až dva listy, kontroluje se po postavení nového, zdali již na tomto stroku není jiný list. Podle toho se poté buď přidá do pomocné tabulky pro světlo nový klíč (*IDStroke*) s hodnotou 1 (je to první list na tomto stroku) nebo se u již stávajícího klíče změní hodnota z 1 na 2 (je to druhý list na tomto stroku).

Pro těžení vzduchu se nepotřebuje žádná pomocná tabulka, a proto se při postavení nové větve přidají všechny jeho stroky jako klíče do zdrojové tabulky s hodnotou rovnou nule.

Samotné těžení zdrojů se provádí v čerpacích cyklech. Při každém tomto cyklu se do jednotlivých stroků, ve kterých se něco těží, načerpá určitý počet zdrojů. Tento počet je specifikován pro každý zdroj zvlášť.

U čerpání zdrojů z půdy se prochází pomocná tabulka *hastTree(1-3)StrokesAreas*. U každého klíče se zjistí jeho hodnota, jež specifikuje oblasti, ze kterých daný stroke čerpá. Pro každou tuto oblast se zjišťují její specifické vlastnosti. Těmito vlastnostmi jsou textura, jakou je vykreslena na mapě, kategorie, do které oblast patří, aktuální a počáteční hodnota, tj. kolik množství zdroje obsahovala oblast na začátku a kolik obsahuje nyní. Aby se nemusela procházet XML struktura pořád dokola a v ní vyhledávat vždy jednu specifickou oblast, jsou tyto vlastnosti uloženy v dalších pomocných tabulkách.

Po zjištění těchto vlastností oblasti zjistíme, jestli je její aktuální hodnota větší než nula. Pokud není, přejdeme na další oblast. Když je větší než nula, tj. dá se z této oblasti těžit, odečteme od této hodnoty hodnotu pro čerpání daného zdroje, tím dostaneme zbývající hodnotu oblasti. Jestliže je zbývající hodnota větší než nula přičteme ve zdrojové tabulce k aktuální hodnotě pro tento stroke hodnotu pro čerpání tohoto zdroje. Jelikož, ale v jednom stroku, může být maximálně 100 z jednoho zdroje. Přičteme tuto hodnotu, jedině pokud bude celková hodnota menší nebo rovna 100. Pokud ale bude větší, přičteme jenom hodnotu, po které bude výsledný obsah stroku 100. Také ještě upravíme hodnotu v pomocné tabulce s aktuální hodnotou oblasti.

V následující ukázce jde vidět čerpání vody u jednoho stroku.

```
local point = hashtable.get ( this.hashTree1StrokesWaterPoint ( ), sIDStroke )
Dischargebility  = this.nDischargebilityWater ( )
if point == nil
then
        hashtable.add ( this.hashTree1StrokesWaterPoint ( ), sIDStroke, Dischargebility )
else
        if point < 100
        then
                point = point + Dischargebility
                if point <= 100
                then
                        hashtable.set ( this.hashTree1StrokesWaterPoint ( ), sIDStroke, point )
                        hashtable.set ( this.hashMapAreaCurrentLevel ( ), sIDArea, leftValue )
                else
```

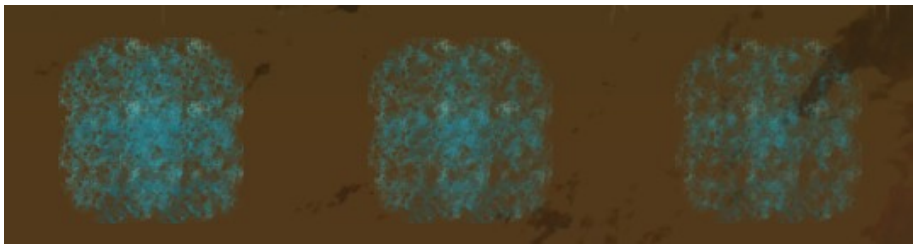
```
                        leftValue = leftValue + (point - 100)
                        hashtable.set ( this.hashTree1StrokesWaterPoint ( ), sIDStroke, 100)
                        hashtable.set ( this.hashMapAreaCurrentLevel ( ), sIDArea, leftValue )
                    end
                end
            end
```

Jelikož se v konkurenčních stromech nezobrazují natěžené zdroje a neexistují pro ně zdrojové tabulky, tak se místo přičtení hodnoty do zdrojové tabulky rovnou přičte energie, kterou tímto čerpáním získal vynásobená koeficientem pro čerpání u konkurenčních stromů. Následný zápis v kódu pro čerpání vody konkurenčního stromu v jednom stroku potom vypadá takto:

```
Dischargebility = Dischargebility * this.nOpponentSuckCoef ( )

this.nTree2EnergyPoints ( this.nTree2EnergyPoints ( ) + (( Dischargebility * this.nWeightWater ( )
)/ this.nAllWeight()))
```

Nově vzniklou aktuální hodnotu porovnáme s počáteční hodnotou. Jakmile je tato nová hodnota vetší než 1/3 a menší než 2/3 počáteční hodnoty a je na tuto oblast použita textura značící její plnou zbývající hodnotu, změníme texturu všech mapových elementů této oblasti na texturu značící střední zbývající hodnotu. Pokud je zbývající hodnota menší než 1/3 a oblast není vykreslena texturou značící nízký zbývající obsah oblasti, použijeme ji na tuto oblast. Takto může hráč na mapě vidět, kolik v dané oblasti ještě zhruba zbývá zdrojů. Ta obrázku Obr. 12 jsou zobrazeny všechny tři textury vody, kde oblast vlevo je plná, oblast uprostřed má okolo poloviny počátečních zdrojů a oblast vpravo má už jen pár posledních zdrojů.



*Obr. 12 – Oblasti vody*

Změna textury se poté provádí tak, že se nejdříve v XML projdou všechny mapové elementy a od těch, které jsou z dané oblasti, se zjistí jejich ID. Podle tohoto ID se poté nalezne objekt na mapě, kterému se jednoduše změní textura.

Následující ukázka kódu ukazuje, jak se mění textura všech mapelemetnů pro danou oblast.

```
if Category == 1
then
```

```
        picture = "water_" .. leftResource .. "_2dmap"
else
        picture = "soil_" .. leftResource .. "_2dmap"
end

for i=0, xml.getElementChildCount ( mapElements ) -1
do
        local mapElement = xml.getElementChildAt ( mapElements, i )
        local hIDarea = xml.getElementAttributeWithName ( mapElement, 'IDArea' )
        local XMLIDArea = string.toNumber ( xml.getAttributeValue ( hIDarea ) )

        local hIDMapElement = xml.getElementAttributeWithName ( mapElement,
'IDMapElement' )
        local IDMapElement = string.toNumber ( xml.getAttributeValue ( hIDMapElement ) )

        local HUDname = this.sHUDMapElementPrefix ( ).. IDMapElement

        if XMLIDArea == IDArea
        then
                local hPicture = hud.getComponent ( this.getUser ( ), HUDname )
                if hPicture ~= nil
                then
                        hud.setComponentBackgroundImage ( hPicture, picture )
                end
        end
end
```

Jakmile je zbývající hodnota menší nebo rovna nule, tj. oblast byla vytěžena, odstraníme tuto oblast z mapy. To se provede stejným způsobem, jako se měnila textura, jenom místo změny textury se daný objekt z mapy odstraní pomocí funkce *hud.destroyComponent (hPicture)*.

Při čerpání světla se prochází pomocná tabulka pro světlo a pro každý její klíč se zjistí jeho hodnota, kolik listů se na daném stroku nachází. Touto hodnotou se poté vynásobí hodnota pro čerpání světla a ta se přičte k hodnotě ve zdrojové tabulce u daného stroku. Díky tomu, že může být maximálně 100 jednoho zdroje v jednom stroku, je zde také kontrola, která přičte hodnotu, po které bude maximální hodnota rovna 100. U vzduchu je to stejné, akorát se rovnou prochází jeho zdrojová tabulka a k hodnotám, uložených u klíčů se přičítá hodnota pro čerpání vzduchu.

K tomu aby byla hra plynulá, neexistují žádné tabulky pro světlo a vzduch u konkurenčních stromů, které by se procházeli a u každé jejich položky by se zvyšovala postupně celková energie stromu. Místo nich je uložen počet listů na stromě a počet stroků, od větví. Poté je při čerpání přidaná energie vynásobena tímto počtem.

Zde je ukázka kódu pro čerpání vzduchu konkurenčního stromu.

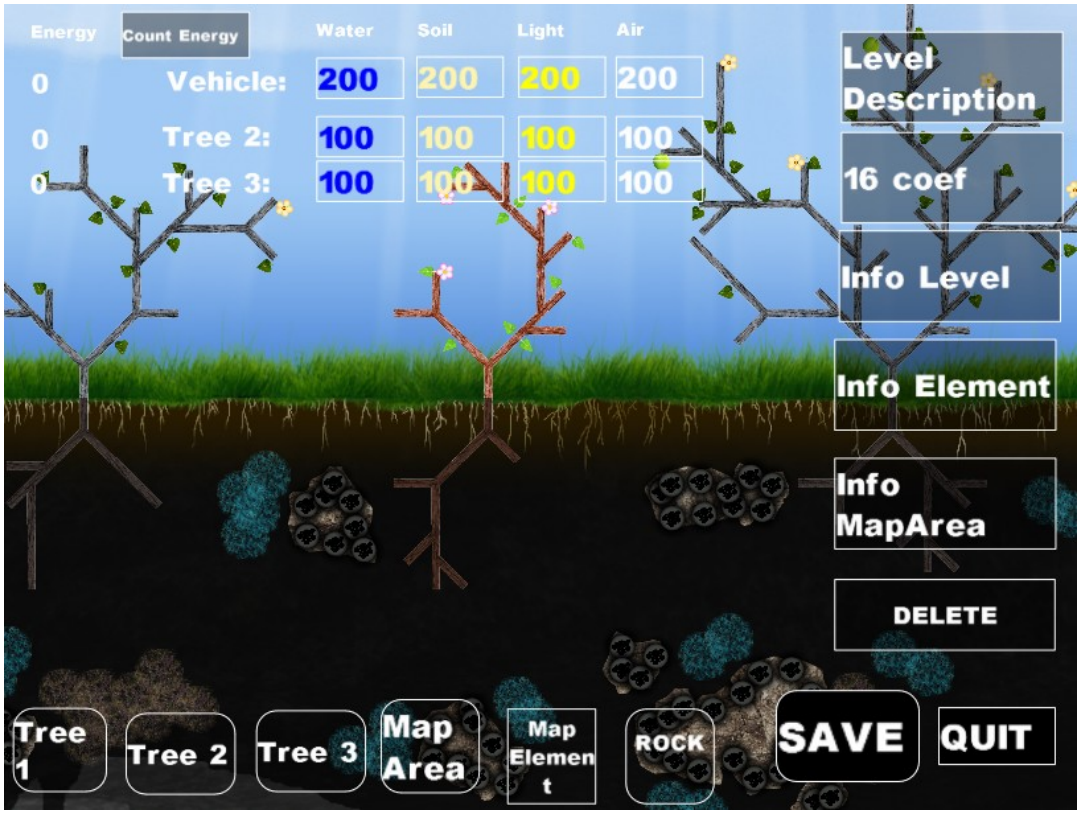
```
local Dischargebility = this.nDischargebilityAir ( )
```

```
Dischargebility = Dischargebility * this.nOpponentSuckCoef ( )
local point =  ( this.nTree2AirStrokes ( ) * Dischargebility )

this.nTree2EnergyPoints ( this.nTree2EnergyPoints ( ) + (( point * this.nWeightAir ( ) )/
this.nAllWeight()))
```

## 4.6 Editor úrovní

Tento nástroj používal herní návrhář k návrhu jednotlivých úrovní hry. Editor byl navržen čistě pro interní účely, čemuž také odpovídá uživatelské rozhraní, jež je zobrazeno na obrázku Obr. 13. V editoru se pracuje čistě ve 2D světě. Na základě specifikování vlastností a postavení stromů a mapelementů se generuje XML soubor. Tento soubor se poté ve hře načítá a odpovídá jedné úrovni.

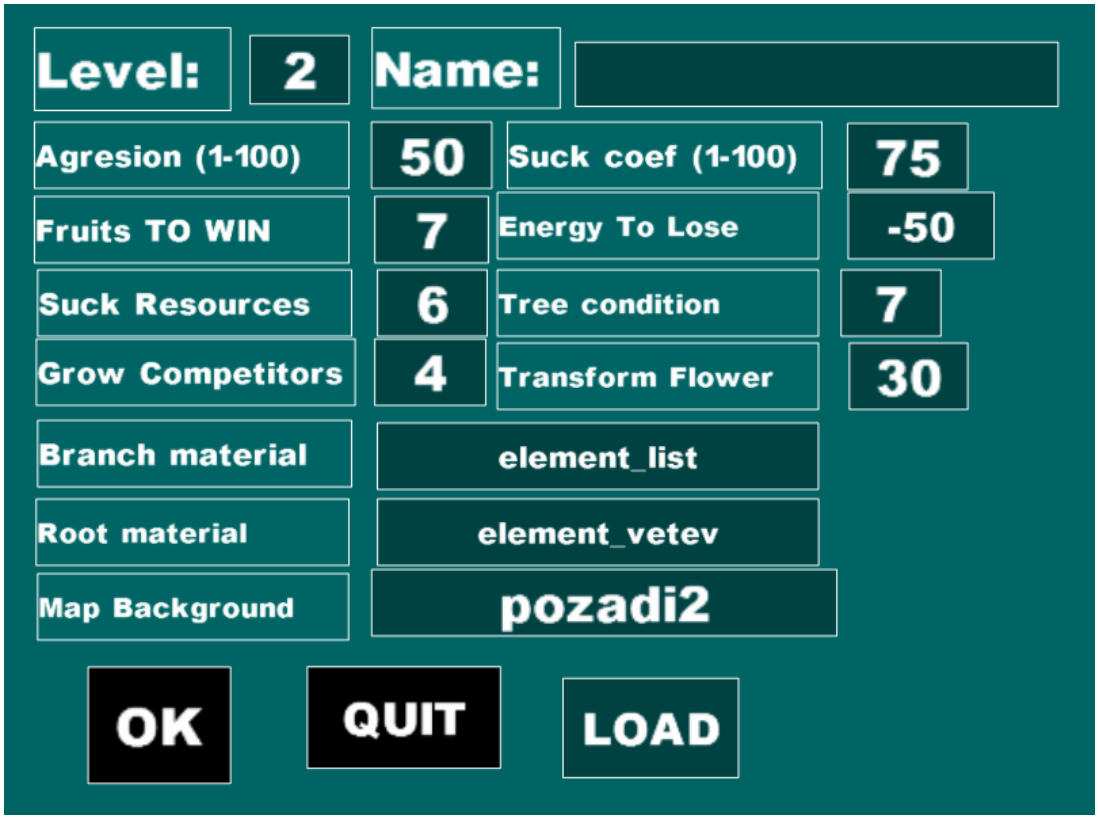



*Obr. 13 – Editor úrovní*

K jeho vytvoření jsem použil HUD editor, ve kterém jsem vytvořil jednotlivé obrazovky s tlačítky a políčky pro vyplnění. Také jsem využil stávající funkce pro vykreslení mapy. Ty jsem trochu upravil, aby jejich funkce odpovídali editoru úrovní.

Nejprve se objeví úvodní obrazovka, zobrazena na obrázku Obr. 14. Je zde možnost vytvoření úplně nové úrovně nebo načtení a následná úprava již vytvořených úrovní. Před vytváření nové úrovně se musí specifikovat číslo úrovně a textura pozadí. Kromě těchto povinných údajů se zde

také nachází prvních pár údajů pro nastavení obtížnosti úrovně. Jsou to všechna časová nastavení, nastavení inteligence konkurenčních stromů a nastavení pro výhru a prohru úrovně. Dále je zde možnost nastavení textur, jež budou poté ve hře použity na větve a kořeny uvnitř stromu. Všechna tato nastavení se dají také ještě změnit přímo při vytváření úrovně, takže se dá například měnit pozadí úrovně a sledovat, které se bude pro danou úroveň nejlépe hodit.




*Obr. 14 – Editor úrovní – úvodní obrazovka*

Po specifikování potřebných údajů se zobrazí prázdná mapa s definovaným pozadím. Na této mapě se poté vytváří celý herní svět. Vytvářejí se zde jednotlivé stromy i s mapelemety a nastavují nastavení pro obtížnost této úrovně. Se zobrazením mapy se také vytvoří XML struktura s elementy *Trees*, *Areas* a *Variables* obsahující elementy pro všechna nastavení obtížnosti úrovně. Vyplněné hodnoty z úvodní obrazovky se uloží do XML struktury, ostatní nastavení pro obtížnost se nastaví na prázdné.

V následujícím kódu je ukázáno, jak se získají a následně zapíší dvě hodnoty z úvodní obrazovky. Je to ID úrovně a počet plodů pro výhru.

```
hComponent = hud.getComponent ( this.getUser ( ), "MissionEditorBuildHUD.EditLevel" )
local level = hud.getLabelText (hComponent )

hComponent = hud.getComponent( this.getUser ( ), "MissionEditorBuildHUD.EditFruitsToWin" )
```

```
local FruitsToWin = hud.getLabelText (hComponent )

local hXMLVariables = xml.appendElementChild ( hXMLLevel, "Variables", "" )
hXMLAtribute = xml.appendElementChild ( hXMLVariables, "IDLevel", level)
hXMLAtribute = xml.appendElementChild ( hXMLVariables, "FruitsToWin", FruitsToWin )
```

Vytváření herního světa pro jednotlivé úrovně funguje na stejném principu jako stavění ve hře. Kromě stavěcího menu pro hráčův strom jsou zde obsažena stavěcí menu pro konkurenční stromy. V každém tomto menu je také možnost postavení plodu. Přítomno je i stavěcí menu pro mapelementy.

Při vytváření jednotlivých stromů se musí nejdříve postavit první větev, na které se pak už budou stavět všechny ostatní (sub)elementy. K tomu aby se dala tato první větev postavit správně, tak aby začínala na rozhraní země a vzduchu, vytvoří se několik možných míst pro její umístění. Větev je pak postavena na to místo ke kterému byla nejblíže, stejně jak je tomu u stavění ve hře. Jelikož na každém herním pozadí je rozhraní mezi zemí a vzduchem přímo uprostřed, odpovídá y-ová souřadnice těchto možných umístění středu obrazovky. X-ové souřadnice jsou rozděleny po skocích 25. Vytváření dalších (sub)elementů funguje na stejném principu jako stavění ve hře. Rozdíl je jen v tom, že nekontroluje dostatečný počet energie a stavět se dá na všech elementech. Může nastat i situace, kdy se objeví 50 možných umístění pro list, pokud bude strom dostatečně velký.

Pokud bychom se při vytváření (sub)elementů uklikli a chtěli nově postavený (sub)element zrušit nebo smazat celý strom a postavit ho úplně jinak, existuje zde tlačítko na mazání. Toto tlačítko aktivuje tzv. mazací mód. Při tomto módu jsou všechna ostatní tlačítka neaktivní. Při vykreslování (sub)elementu na mapě se každému (sub)elementu přiřadí tři akce. V každé akci se definuje handler s určitými parametry, který bude zavolán. Poté se každá akce přiradí k aktivitě, při které se vykoná. U první je to kliknutí na daný objekt, u dalších dvou, když myš přejede na daný objekt a když z něj odjede. Tyto akce jsou neaktivní po celou dobu stavění, aktivují se až v mazacím módu.

V následující ukázce je ukázáno nastavení akce pro kliknutí a její následné přiřazení. Také je zde ukázáno přiřazení akcí pro přejetí a odjetí myši nad daným obrázkem.

```
local hAction = hud.newAction ( this.getUser ( ), "myAction"..HUDname )

hud.beginActionCommand ( hAction, hud.kCommandTypeSendEventToUser )
hud.pushActionCommandRuntimeArgument ( hAction, hud.kRuntimeValueCurrentUser )
hud.pushActionCommandArgument ( hAction, "MissionEditorAI" )
hud.pushActionCommandArgument ( hAction, "onMapItemClick" )
hud.pushActionCommandArgument ( hAction, HUDname )
hud.pushActionCommandArgument ( hAction, IdElement )
hud.pushActionCommandArgument ( hAction, IdTree )
hud.pushActionCommandArgument ( hAction, TypeElement)
```

```
hud.endActionCommand ( hAction )

hud.setButtonOnClickAction ( hPicture, hAction )
hud.addComponentEventHandler ( hPicture, hud.kEventTypeMouseEnter, hActionEnter )
hud.addComponentEventHandler ( hPicture, hud.kEventTypeMouseLeave, hActionLeave )
```

U handlerů pro přejetí a odjetí myši se jenom mění textura daného (sub)elementu. V handleru pro kliknutí se ale kromě jednoduchého odstranění objektu z mapy, musí také odstranit daný (sub)element z XML struktury. U subelementu je to jenom nalezení daného subelementu v XML a jeho odstranění pomocí funkce *xml.removeElementChild (hXMLElement, hChild)*.

U elementů se musí zachovat pořadí jeho potomků, tak že jeho první potomek má startovní souřadnice rovny prvním koncovým souřadnicím (EndX1, EndY1) jeho rodiče. Druhý je má rovny druhým koncovým souřadnicím a třetí třetím. Díky tomuto zachování se poté správně generují šipky uvnitř stromu. Aby toto pořadí bylo zachováno, nelze čistě smazat daný element a jeho rodičovi přidat nový zátkový element se souřadnicemi jeho smazaného potomka. Pokud bychom takhle smazali například prvního potomka a přidali nový zátkový element, bude tento nový zátkový element jako poslední potomek. Potom při postavení nového elementu na tento zátkový bude onen element jako poslední potomek, místo aby byl první. Proto se při mazání elementů, neodstraňuje přímo onen element, ale odstraňují se všichni jeho potomci a v tomto elementu se poté upraví všechny jeho atributy, aby odpovídali zátkovému elementu.

Jelikož se každý element skládá ze stroků, musí se i ony odstranit. To se provede stejně jako u subelementů, akorát se vyhledají a smažou všechny stroky, kteří náleží danému elementu.

Protože se při smazání elementu, který obsahuje ještě několik dalších elementů, odstraní i tyto další elementy z XML, ale na mapě zůstanou zobrazeny a jejich stroky také zůstanou v XML, musí se z mapy také smazat, tak jako se smazal jejich rodič. Aby se předešlo situaci, kdy by byl smazán nějaký element s potomky z XML, ale jeho potomci by stále byli zobrazeni na mapě, nelze ukončit mazací mód, dokud nejsou odstraněni i jeho potomci. Při ukončování mazacího módu se kontroluje, zda existuje nějaký stroke, jenž náleží elementu, který se v XML nenachází. Pokud takový existuje, znamená to, že nebyli smazáni všichni potomci a uživatel je o tom informován. Mazací mód také nelze ukončit, pokud existuje nějaký subelement, jehož startovní souřadnice se nerovnají koncovým souřadnicím nějakého stroku.

Mazání je také možné u mapelementů a funguje na stejném principu jako mazání (sub)elementů, akorát se zde neprovádí žádná kontrola při ukončení mazacího módu.

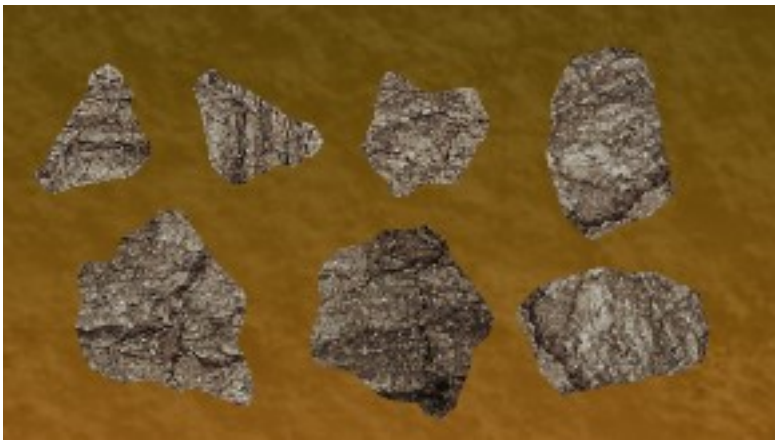
U vytváření zdrojů v půdě se nejdříve musí specifikovat oblast. Musí se určit kategorie, zda se bude jednat o vodu nebo půdu a nastavit její počáteční hodnotu. Je zde také nepovinný atribut popis oblasti. Když se to tohle nastaví, objeví se tlačítko pro vytvoření jednotlivých mapelemetů této oblasti. Po kliknutí na toto tlačítko a přetažení myši se na místo přetažení postaví daný mapelement. Princip je stejný jako u stavění (sub)elemetů, akorát se nevytvářejí možná místo pro

postavení a mapelement lze umístit kdekoliv na mapě. Jeho souřadnice se rovnají souřadicím přetažení. Jedna oblast se může skládat z více mapelementů, jejichž maximální počet není omezený. Počet oblastí také není omezený. Tlačítko pro stavění mapelemetů je aktuální k poslední specifikované oblasti.

K tomu aby bylo možné jednotlivé oblasti ještě upravovat, existuje zde tlačítko *Info MapArea*, které zapne editační mód pro oblasti. Při jeho aktivaci se aktivují akce u mapelementů, jež mění jejich texturu po přejetí myší nad nimi a po kliknutí se zobrazí editační okno, ve kterém lze změnit počáteční hodnotu a popis. Kategorie změnit nelze. Hodnoty do editačního okna se získávají tak, že akce pro kliknutí předává parametr *IDArea*, podle kterého se v XML najde příslušná oblast. Z ní se načtou hodnoty atributů, které se zobrazí v editačním okně.

V editačním okně se ještě nachází tlačítko pro přidání dalšího mapelementu pro tuto oblast. Po jeho stisknutí se nastaví tlačítko pro přidání mapelementu na naši upravovanou oblast a vybranou oblast lze tak lehce rozšiřovat.

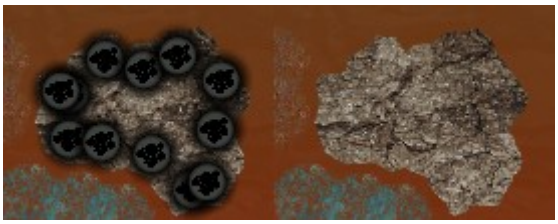
Kromě zdrojů v půdě se mohou vytvářet i skály. K tomu slouží další stavěcí menu. V tomto menu je na výběr celkem sedm typů skal, jejichž spojováním se dají vytvořit skály všech možných velikostí i tvarů. Všechny typy jsou zobrazeny na obrázku Obr. 15.



*Obr. 15 - Typy skal*

Když si pomocí těchto skal vytvoříme jednu velkou skálu, samotné počítání kolizí kořenů s touto skálou by bylo náročné. Proto existuje ještě jeden speciální typ skály. Tento typ se už ale ve hře vůbec nezobrazuje a slouží jenom pro přesné počítaní kolizí. Je to takzvaná neviditelná skála. Tato neviditelná skála je menší než ostatní typy skal a je kulatého tvaru. Umisťuje se na okraje normálních skal. Tyto neviditelné skály se poté přidávají ve hře do mřížky mapy a určují místa, přes které nelze stavět.

Na obrázku Obr. 16 je zobrazena skála obehnána neviditelnou skálou, v levé části je zobrazena v editoru a v pravé části je ta stejná skála zobrazena přímo ve hře.



*Obr. 16 – Neviditelná sklála*

Mimo vytváření herního světa se zde také nastavují všechny parametry pro obtížnost úrovně. Pro každý parametr zde existuje jedno editační políčko. Text z tohoto políčka se vezme a uloží do XML do patřičného elementu. Místo počáteční energie stromu se zde nastavují počáteční hodnoty jednotlivých zdrojů, ty se poté při ukládání přepočtou podle nastavených vah na energii. K tomu aby možno vidět, kolik budou mít jednotlivé stromy energie, slouží tlačítko *Count Energy*, jež tyto energie vypočte a zobrazí.

Při ukládání úrovně se zkontrolují všechny stroky, jestli některé z nich netěží nějaký zdroj a podle toho se naplní jednotlivé pomocné tabulky pro těžení. Tyto pomocné tabulky se poté vloží do XML struktury. Po této kontrole se vytvoří nový XML soubor nebo se přepíše již stávající. Jméno souboru je *LevelXX.xml*, kde XX je číslo úrovně. Vytvoření takového souboru se provede následujícím příkazem:

```
xml.send ( this.Level ( ), "file://"..application.getPackDirectory ( )..."/Level"..this.sIDLevel (
)..".xml" )
```

Ve hře se pak tyto soubory načítají a tvoří jednotlivé úrovně. Načítání se provádí tímto kódem:

```
local sUrl = "file://"..application.getPackDirectory ( )..."/Level"..nLevel .. ".xml"
xml.receive ( this.Level ( ), sUrl )
```

# 5 Závěr

Prošel jsem si celým vývojovým cyklem hry od specifikace záměru přes návrh a implementaci až po samotnou distribuci do on-line obchodu Apple App Store. Při realizaci hry jsem měl možnost vyvíjet a navrhovat klíčové herní algoritmy a ověřovat si vazbu mezi teorií a praktickou implementaci, zejména průchodnosti algoritmu na cílové hardwarové architektuře. Jako pozitivní hodnotím seznámení s novým vývojovým prostředím a programovacím jazykem Lua. Byl jsem osobně překvapen, jakých výsledků lze dosáhnout i takto poměrně skromným jazykem třetí generace.

Vzhledem k tomu, že distribuce hry proběhla nedávno a marketing se rozjížděl, nemůžu popsat zkušenost s komerčním uplatněním hry. Hra byla primárně určena pro anglicky hovořící trh, mezi hlavní cílové země patří USA a Velká Britanie. Presto již máme výsledky první recenze nezávislého herního serveru Crazy Mike's apps [6].

Díky úkolům, jež jsem dostal na starost a které se většinou zabývali herní logikou a prací s 2D objekty jsem tak mohl vyzkoušet vytvoření umělé inteligence a získal jsem další znalosti pro zpracování grafiky na mobilních platformách. Také jsem si vyzkoušel samotnou distribuci do on-line obchodu Apple App Store [1].

Dalším krokem bude umístění hry do obchodu Google Play. Do něj však do 12. 4. 2012 nebylo možné vkládat komerční aplikace z území České Republiky [3].

# 6 Použitá literatura

[1] App Store - Re!Sources. *Apple* [online]. [cit. 2012-04-25]. Dostupné z: http://itunes.apple.com/gb/app/re!sources/id508545253.

[2] HAVRLANT, Lukáš. Vektory. *Matematika polopatě* [online]. [cit. 2012-04-25]. Dostupné z: http://www.matweb.cz/vektory.

[3] HRÁČEK, Filip. Čeští vývojáři teď mohou prodávat aplikace pro Android. *Google blog ČR* [online]. [cit. 2012-04-25]. Dostupné z: http://google-cz.blogspot.com/2012/04/cesti-vyvojari-ted-mohou-prodavat.html.

[4] PILATO, C, Ben COLLINS-SUSSMAN a Brian FITZPATRICK. *Version control with subversion*. 2nd ed. Beijing: O'Reilly, 2008. ISBN 978-0-596-51033-6.

[5] Počítačová hra. In: *Wikipedia*: *the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 23. 4. 2012 [cit. 2012-04-25]. Dostupné z: http://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%A1_hra.

[6] Re!Sources - Super Challenging Earth-Friendly Game!. *Crazy Mike's apps* [online]. [cit. 2012-04-25]. Dostupné z: http://www.crazymikesapps.com/resources-super-challenging-earth-friendly-game/.

[7] *Shiva 3D* [online]. © 2003 - 2012 [cit. 2012-04-25]. Dostupné z: http://www.stonetrip.com.

[8] SOUČEK, Eduard. *Statistika Pro Ekonomy*. Praha: VŠEM, 2007. ISBN 978-80-86730-06-6.

[9] *The Programming Language Lua* [online]. [17 Apr 2012] [cit. 2012-04-25]. Dostupné z: http://www.lua.org/.

[10] *UNITY* [online]. © 2012 [cit. 2012-04-25]. Dostupné z: http://unity3d.com/unity/.

[11] *Unreal Engine* [online]. © 2008-2012 [cit. 2012-04-25]. Dostupné z: http://www.unrealengine.com/.

[12] VOJÁČEK, Jakub. Sinová a kosinová věta. *Matematika pro každého* [online]. [cit. 2012-04-25]. Dostupné z: http://maths.cz/clanky/sinova-a-kosinova-veta.html.